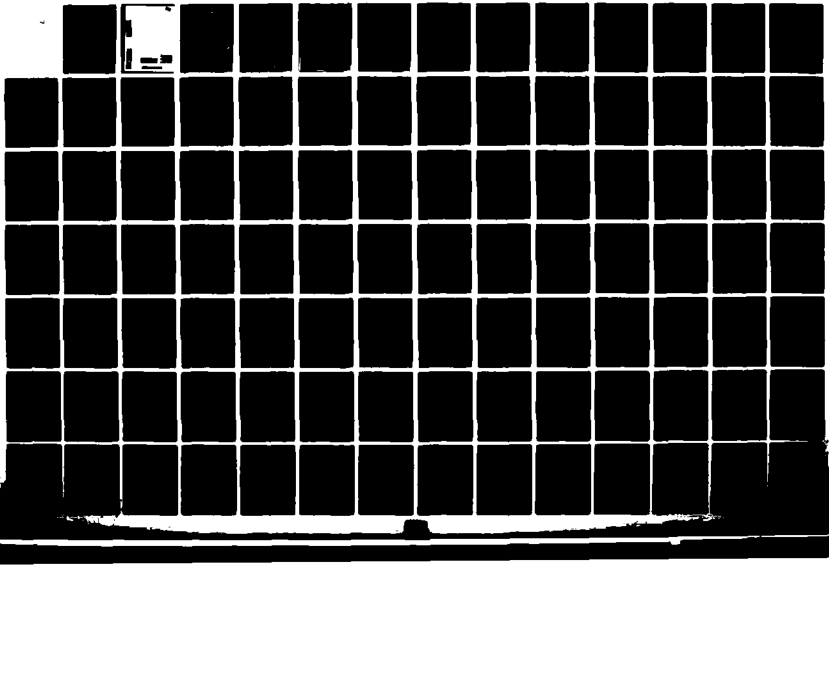
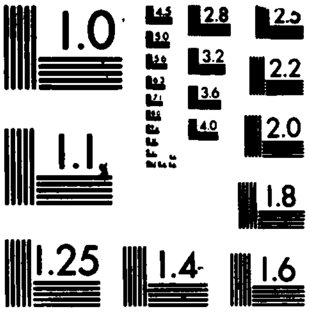
AD-A142 018 NAMING IN A PROGRAMMING SUPPORT ENVIRONMENT(U) 1/2
MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER
SCIENCE J N LANCASTER FEB 84 MIT/LCS/TR-312
UNCLASSIFIED N00014-75-C-0661 F/G 9/2 NL




1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(13)

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-312

NAMING IN A PROGRAMMING SUPPORT ENVIRONMENT

Julia Nan Lancaster

[illegible] was supported in part by the Defense Advanced Research [illegible] Agency of the Department of Defense and was [illegible] by the [illegible] of Naval Research under Contract No. N000 [illegible]

[illegible] SQUARE, CAMBRIDGE, MASSACHUSETTS [illegible]

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER MIT/LCS/TR-312	2. GOVT ACCESSION NO. AH2018	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Naming In A Programming Support Environment		5. TYPE OF REPORT & PERIOD COVERED M.S. Thesis February 1984
		6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-312
7. AUTHOR(*s*) Julia Nan Lancaster		8. CONTRACT OR GRANT NUMBER(*s*) DARPA/DOD N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/DOD 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE February 1984
		13. NUMBER OF PAGES 169
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) ONR/Department of the Navy Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. (*of this report*) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for Public Release, distribution is unlimited

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Software version control, configuration identification, configuration management, naming, programming environments

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

Modular programming supports the decomposition of large programs into subtasks called modules. Although any two implementations of a module must provide identical interfaces and generally the same behavior, they may differ in subtle and sometimes significant ways. This thesis addresses the problem of identifying the multiple implementations of a module. We propose a naming system based on attributes that allows users to

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

20. continued

express their insights about the properties of and differences between implementations. We then develop two mechanisms using our naming system to retrieve implementations from a software library. One retrieves an implementation of an individual module; the other retrieves implementations of each of the modules comprising a program.

Naming In A Programming Support Environment

by

Julia Nan Lancaster

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Naming In A Programming Support Environment

by

Julia Nan Lancaster

Abstract

Modular programming supports the decomposition of large programs into subtasks called modules. Although any two implementations of a module must provide identical interfaces and generally the same behavior, ~~they~~ these may differ in subtle and sometimes significant ways. This thesis addresses the problem of identifying the multiple implementations of a module. The author ~~We~~ proposes a naming system based on attributes that allows users to express their insights about the properties of and differences between implementations. He We then develops two mechanisms using ~~our~~ this naming system to retrieve implementations from a software library. One retrieves an implementation of an individual module; the other retrieves implementations of each of the modules comprising a program.

Keywords: software version control, configuration identification, configuration management, naming, programming environments

This report is a revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on August 26, 1983 in partial fulfillment of the requirements for the Degree of Master of Science.

Thesis Supervisor: Barbara H. Liskov
Title: Professor of Computer Science

Acknowledgments

I would like to express my appreciation to my advisor, Professor Barbara Liskov, for her insights, suggestions, criticism, and editing. Without her help, this thesis would not have been possible. I would also like to thank the members of the Programming Methodology, Systematic Program Development, and Theory of Distributed Computing Groups, for their friendship and for listening patiently to my half-formed ideas. Finally, I owe special thanks to Janie Weinberg and Randy Forgaard for the encouragement and technical support they provided in the final stages of this research.

Table of Contents

Table of Figures

Chapter One

Introduction

Modular programming supports the decomposition of large programs into subtasks called modules [28]. Each module corresponds to "a black box" with a generally agreed upon behavior. A module is defined by a specification (either formal or informal) giving its interface and externally visible behavior. A module may have many implementations, each of which must satisfy the module's specification. An implementation provides the mechanism by which the module's task is achieved; it offers an internal view of the black box. This approach to program construction permits the implementations for distinct modules to be developed independently because the modules' specifications are fixed.

In theory, each of a module's implementations may be substituted for any of the others. Because they satisfy the same specification, they should all be equivalent. For two reasons, however, this may not be true. First, some implementations may contain bugs or other errors, and may not exhibit the correct external behavior. They fail to satisfy the module's specification. Second, the module's specification does not describe the complete behavior of its implementations. Two implementations satisfying the same specification may differ in small but potentially significant ways. For example, they could use different algorithms, support different levels of metering and statistics-gathering, have different performance characteristics, or run on different target configurations. The specification may not address such "micro-behaviors."

To identify the implementation of a module that best meets his needs, the user must evaluate the complete behavior of all the module's implementations. The externally visible behavior described by the specification is not sufficient. The user selects from the module's implementations based on their correctness with reference to the specification, and on any micro-behaviors he considers significant. Thus, implementation identification requires the user to have a complete understanding of the properties of and differences between implementations.

Two factors make implementation identification difficult. The first is the sheer volume of information. For a large program composed of many modules, each of which has multiple implementations, the number of implementations may be staggering. The user cannot hope to comprehend all of them. In addition, several distinct aspects of an implementation may be of interest in different circumstances. For example, one user might be concerned about an *implementation's algorithm, while another focuses on its performance characteristics.* Multiple significant features per implementation add to the complexity by increasing the amount of information the user must retain.

The second factor is sharing of modules. A large program is often developed by a team of programmers, each of which is responsible for only a subset of a program's modules. A programmer shares the modules created by his teammates, but is unlikely to fully understand the distinctions between implementations of modules not under his control.

In this thesis, we propose a naming scheme to assist the user with implementation identification. Names provide a means for users to refer to the implementations of a module. Our approach allows users to express their insights about implementations as names. An implementation's names record information about its properties. Names that reflect users' understandings of the behavior of implementations document implementations' micro-behaviors and help preserve the user's comprehension of the differences between implementations. They help the user retain and organize the large volume of information relevant to implementation identification. Expressive names also serve as communication media among sharers of a module, identifying the significant features of and differences between the module's implementations.

1.1 Overview of the Research

Ours is a descriptive approach to implementation identification. The names of an implementation document its features and behavior. A user selects an implementation by describing the behavior he would like, and then choosing an implementation whose names indicate that it provides the desired behavior.

The following scenario demonstrates how expressive names may be used. Many distinct properties may be useful to distinguish among the implementations of a module: algorithm, performance, correctness, and so on. When a user determines that an aspect of an implementation is a significant factor in understanding its behavior, he records that property as one of the implementation's names. Expressive names associate a description of its behavior with the each implementation. To select an implementation, a user identifies the behavior he wants and formulates a description in terms of specific properties. The description acts as a complex form of name, identifying an implementation whose names match the properties required by the user.

This thesis is divided into four main parts. First, we analyze the problem of implementation identification to determine what characteristics of a naming system can support a descriptive approach as outlined above. We find six requirements on naming systems that contribute to implementation identification.

- **Expressiveness.** Names should have the capacity to express a wide variety of information in a form meaningful to people.

- **Multiplicity**. The naming scheme should support multiple names per implementation.

- **Non-uniqueness**. Non-unique names, referring to more than one implementation, should be supported.

- **Independence**. The naming system should allow users to treat names independently, without considering either other names for the same implementation or names for other implementations.

- **Commonality**. The naming system should recognize and highlight commonality occurring when two names reflect a single class of information.

- **Mutability**. The naming system should support dynamic name creation and modification.

Second, we propose a naming system based on attributes that satisfies the requirements articulated above. Each name in our scheme consists of a pair: an attribute name and its value. The attribute name represents a class of information; the value determines the instance of the class. The attribute name expresses the meaning of the value, and exposes commonality among the names of multiple implementations. Users may create or modify names as necessary by specifying both the attribute names and values to be associated with an implementation.

The remaining parts of the thesis explore two retrieval mechanisms based on user-provided descriptions of the desired behavior. The first mechanism retrieves an implementation for a single module given a description of its properties. A description is represented by conditions on attributes. Retrieval is a process of searching for an implementation whose attribute values satisfy the conditions specified by the description. We offer a proposal for this simple kind of retrieval modeled after relational database queries.

A second retrieval problem consists of selecting implementations for each of the modules in a program. This form of retrieval is significantly more complex for two reasons.

First, it may involve sets of modules whose properties of interest are disjoint. Second, it is likely to involve the work of several programmers, necessitating the sharing of modules whose properties may not be well understood. These two factors combine to make writing a description of the desired program behavior a very difficult task. We do not offer a finely-tuned solution to this problem. Although we do suggest extensions to the mechanism for the single module retrieval as a first cut at a solution for the multiple module case, our focus in this area is expose the issues making programming composition a difficult problem.

1.2 Related Work

Research related to this thesis can be divided into two areas, the naming of implementations in a programming support environment, and program composition, the process of binding together implementations of several modules to make a complete program. We first look at common practice in these two areas, then consider recent research in each area.

In recent years, many programming support environments have been proposed. But although each has a mechanism for naming implementations, very few have suggested names designed to support implementation identification. Most use very simple naming schemes. Three common approaches are represented by the Microprocessor Software Control Facility (MSEF) [17], the Source Code Control System (SCCS) [22, 30, 36], and the Ada Language System (ALS) [32]. MSEF uses a hierarchical name space in which two implementations of the same module share the same name, but are differentiated by version numbers. SCCS also uses multi-level names for implementations. The first level identifies the name of the module; successive levels are numbers. A common convention defines the second level to be the number of the most recent release, and the third to be the number of small revisions since the last release. The ALS uses three-level names, with the module name again as the first level. The second level distinguishes major alternatives in implementation, such as between implementations using different algorithms. The third enumerates small revisions in the history of an alternative's development.

Current practice in program composition relies primarily on ad hoc mechanisms. In general, when it is time to compose a program, each of the programmers collaborating on the program submit the names of their "best" implementation for each of the modules they are responsible for. We consider this chaotic approach unacceptable.

Few papers exist discussing the naming of implementations and its impact on implementation identification. Huff [21] presents an analysis of naming issues supporting implementation identification. Although we agree with her analysis as far as it goes, her focus is on machine-derivable and controllable information. She recognizes other distinctions between implementations, such as algorithms and performance properties, but does not analyze them because they cannot be detected by machine. The naming scheme she proposes relies mainly on history and derivation information, and consequently provides minimal support for implementation identification.

Cheatham [10, 11, 29] performs a similar analysis. Because he also focuses on machine-derivable information, his analysis suffers from the same defects as Huff's. In contrast, however, he proposes a more versatile naming scheme based on partitionings. A partitioning divides a set of implementations into mutually exclusive subsets called partitions. It has a name, and each of its partitions also has a name. Each implementation has associated with it a set of partitioning name - partition name pairs, identifying the partitions to which it belongs. The user selecting an implementation specifies the module name, a set of partitioning memberships, and a version number. This approach allows the user to express the non-machine-derivable information as names in the form of partitioning-partition pairs. The chief drawback of this proposal is that relies on a simplistic retrieval mechanism using string pattern matching. Our research extends Cheatham's approach to develop a more sophisticated naming scheme and retrieval mechanism.

Research on program composition can be divided into two general forms. The first uses a specialized form of command file describing a program. The description enumerates the program's component implementations and contains commands for building the program by compiling, linking, etc., its components. A special tool interprets the description file and invokes the commands. The Unix Make [14, 18, 36] and the Lisp Machine Makesystem [37] facilities are examples of this approach.

Make uses a specialized command file called the "Makefile", containing a description of a program in the form of a tree. Each node represents a component in the program's development, for example, a source version of an implementation of one of the program's modules, or an executable module formed by linking together the results of compiling source versions of several modules. A node's children are the components it depends upon. Associated with each node is a "make rule" that supplies the command sequence for building the node or re-making the node when any of its children nodes change. For example, a make rule might specify that a node representing an executable module be built by linking together its children nodes.

The Lisp Machine Makesystem command provides a very similar facility. As with the Unix Make facility, Makesystem employs a program description enumerating the implementations comprising a program along with commands for building the program from its components. Makesystem interprets the description and performs the specified actions.

The second approach to composition is based on the work of DeRemer and Kron [16].

They argue that program development consists of two separable subtasks that they called "programming in the large" and "programming in the small." Programming in the large defines and describes the structure of programs at the level of modules; programming in the small is the lower level task of implementing individual modules. They propose a new class of languages, called module interconnection languages, for programming in the large. Many proposals for program composition are extensions of this basic idea. The work of Thomas [33], Tichy [34, 35], Schmidt and Lampson [23, 24, 31], Cooprider [13], Archibald [1], and the members of the Gandalf [19, 20] and Mesa [27] projects fall within this category.

DeRemer's and Kron's original module interconnection language and all its successors share a common approach and many basic concepts. A module description defines the interface of a module, listing the resources (such as procedures or data objects) it provides for others and the resources it requires of others. A system description specifies a composition of modules and other systems, representing a complete program. A user builds a program by supplying an implementation for each of the modules contained in the system description. The amount of automated support provided by these proposals varies.

We distinguish two subtasks of program composition: identifying the modules comprising a program, and selecting implementations of each of them. Our focus is on the second of the two subtasks. In contrast, the two command file approaches described above do not distinguish the two tasks at all. The special command files enumerate a program's components to the implementation level, not at an abstract module level. The module interconnection language proposals, on the other hand, do make the distinction, but focus on the first of the two subtasks.

The work that appears to be most closely related to ours is Cargill's [7, 8] research on organizing a family of programs in a file system. A family is a collection of closely related programs, identical in all but a few modules. For example, several versions of a portable operating system targeted to different machines constitute a family. Cargill uses a file system to store program components. He stores a module having one implementation shared by all family members as a file. Modules for which different family members require different implementations are stored as a directory whose entries are the implementations. Each of these implementations is named by the family member to which it belongs. Cargill proposes a tool that searches the file system and retrieves all the implementations belonging to a family member specified by the user.

Although Cargill's naming scheme is less sophisticated than ours, his intention is to allow users to select a set of implementations with a specific property (which, in his case, consists of membership in a particular family member's components). His approach, however, is limited because only one name is permitted per implementation. This limitation constrains the user to express only one property of the implementation. In addition, Cargill's method cannot accommodate names that are not unique across the implementations of a module.

1.3 Organization of the Thesis

We use the programming language CLU [25, 26] and the CLU programming support environment as a basis for our work. Chapter 2 provides background information on the CLU programming support environment, focusing in particular on a centralized software database called the Library. It also introduces some details of CLU and CLU syntax used in examples throughout the thesis. Chapter 3 contains an analysis of naming issues, identifying six characteristics of a naming system that support implementation identification. A proposal for a naming scheme that satisfies these requirements is contained in Chapter 4. Chapters 5 and 6 present the two retrieval mechanisms for implementations. An evaluation of our work and suggestions for further investigations are found in Chapter 7.

Chapter Two

Background

In this chapter we describe the CLU programming environment and the program development process using CLU [25, 26]. CLU is a strongly-typed programming language supporting modular construction of programs. A CLU program is divided into a number of modules, each of which contributes a part of the program's function. Individual modules are developed and compiled separately, with the CLU programming support environment performing inter-module type checking. Before a program is run, the compiled forms of its modules are combined into an executable form.

Each module in a program implements an abstraction. CLU supports three kinds of abstractions: procedural, control, and data. Procedural abstractions compute a set of results from a set of arguments. Control abstractions compute a series of sets of results from a set of arguments. They are used in conjunction with CLU control statements to perform repetitive computations. Data abstractions define a new type of data object and also provide a set of operations for manipulating the objects. From outside the abstraction, objects of the new type may be accessed only through the provided operations. CLU also supports parameterized abstractions which represent a whole class of abstractions. Parameterized abstractions are not addressed by this thesis.

The CLU programming support environment is a collection of tools centered around a software database called the Library. The Library contains all the information known about existing abstractions. Library commands allow the user to create new abstractions or programs, and to modify or retrieve information about existing ones. In addition, all tools in the CLU programming support environment interact with the Library.

The chapter is organized as follows. Section 2.1 discusses the Library and its contents. Section 2.2 traces program development using CLU and the Library. Our model of how an abstraction evolves and new implementations are developed is presented in Section 2.3. Finally, Section 2.4 introduces some notation used in examples throughout the thesis.

2.1 The CLU Library

Each abstraction is represented in the Library by a *description unit*, or *du*, which contains all information relating to the abstraction. Although a variety of information may be stored in a du, the only required component is the abstraction's interface specification. Other optional components may include formal specifications, implementations, executable programs (called compositions in the Library), documentation, test programs and results, status information, and so on. Because of their important roles in program development, the interface specification, *implementations*, and compositions are the focus of concern in this thesis and are described in greater detail below. Figure 2-1 shows a simplified view of a du, containing only these three kinds of components.

The interface specification defines the abstraction's external interface and provides the information needed to support type-checking. It is represented in the Library as

```
Specification =    Interface signature
                   Set of abstraction bindings.
```

For procedural and control abstractions, the interface signature names the the number and types of the arguments and results, as well as the names of any exceptional return conditions and the number and types of results associated with each exception. For example,

proctype (T1) **returns** (T2, T3) **signals** (ExceptB (T4))

is the interface signature of a procedural abstraction taking one argument of type T1, returning two results of type T2 and T3, and having an exceptional return condition named

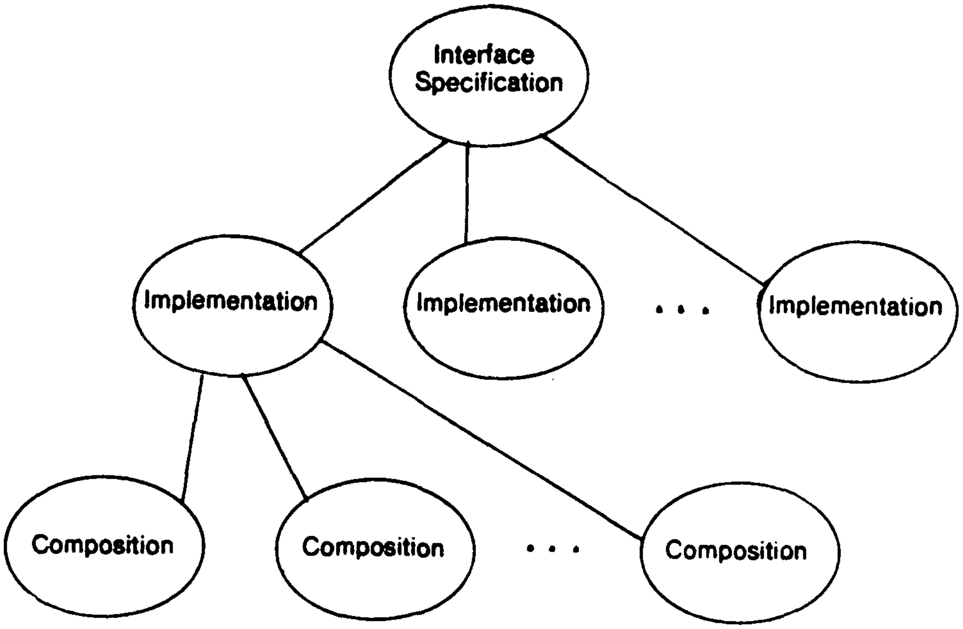


Figure 2-1:A Simplified View of a DU

ExceptB with a result of type T4. The interface signature of a data abstraction consists of the interface signatures of each of the abstraction's operations.

The abstraction bindings map the names of abstractions used in the interface signature to dus in the library representing those abstractions. Thus, in the example above the abstraction bindings would bind dus to the names T1, T2, T3, and T4.

A du may contain an arbitrary number of implementations. Implementations provide an inside view of the abstraction, supplying the mechanism by which the external behavior of the abstraction is achieved. An implementation's representation in the Library has three parts:

Implementation =	Source module Binary module Set of abstraction bindings.

The source module is the CLU source code version of the implementation. The binary module contains the machine language version produced by compilation, along with information about external references needed by the linker. The bindings are sets of mappings resolving the names of subsidiary abstractions used in the source module. An implementation's abstraction bindings record abstractions used in addition to those used in the interface. For example, a dictionary example could be implemented in terms of a hashing abstraction. The hashing abstraction does not appear in the dictionary abstraction's interface, but implementations of the dictionary invoke operations provided by the hashing abstraction.

A composition represents an executable form of the abstraction, formed by combining an implementation of the abstraction with implementations of subsidiary abstractions. A du may contain zero or more compositions. Each is represented as follows:

Composition = Set of component implementations
Executable module.

The set of component implementations contains an implementation for each of the abstractions comprising the program. The CLU linker combines these implementations to produce the executable module. The executable module is the executable program form of the abstraction that can be run by the user.

The Library itself is organized as a sea of dus. Each du is referred to by a unique identifier. We assume that the CLU programming environment provides a more user-friendly name space, but do not address that issue in this thesis.

2.2 Program Development Using the Library

CLU supports modular construction of programs. Instead of being a single monolithic unit, a CLU program is decomposed into multiple abstractions that cooperate to perform the program's function. In general, at the highest level of abstraction, a program has a single abstraction representing the whole program. The implementation of this abstraction invokes implementations of others to perform subsidiary tasks. These may in turn invoke others, and so on.

A common device for describing the structure of a CLU program is the *dependency graph*. A dependency in CLU is a relationship from an implementation to a subsidiary abstraction. We say an implementation of abstraction A *depends* on abstraction B if the implementation of A uses B to perform a subsidiary task. A dependency graph is a directed graph whose nodes correspond to implementations and whose arcs record dependency relations. The parent implementation (at the tail of the arc) depends on the abstractions implemented by its children. Cycles in the graph indicate recursion. A node with no parents implements a top-level abstraction.

Although it is occasionally useful to build a program with more than one top-level abstraction, in this thesis we address only simple programs with a single top-level abstraction. It is possible, however, to generalize all results in this thesis to more complex programs with multiple top-level abstractions.

In developing a new program, the user first determines the abstractions that will comprise it. He creates dus to represent each of these abstractions in the Library, supplying the interface specifications as part of the du-creation process. Implementations can be created and compiled as necessary. Development of implementations of distinct abstractions proceeds independently, in any order. When implementations exist and have been compiled for all the abstractions in the program's dependency graph, the user builds a composition for the top-level abstraction, selecting implementations for each of the abstractions in the program and binding them together into an executable module.

2.2.1 Defining a New Abstraction

When a user invents a new abstraction, he creates a du for it in the Library. As part of du-creation, he must supply its interface specification, both the interface signature and the bindings between used abstractions and the dus that represent them. This defines the abstraction to the outside world and allows it to be referred to by other abstractions. Note that since the interface specification contains all information necessary to support type-checking, an implementation referring to an abstraction may be compiled and fully type-checked as soon as the abstraction's du is created, regardless of whether or not any implementations of the abstraction exist.

The interface specification of an abstraction is not permitted to change. CLU supports inter-module type checking by verifying at compilation that an implementation's usage of a subsidiary abstraction matches the interface specification in the subsidiary's du. If the interface were subsequently modified, type checking would be invalidated. Consequently, the Library prohibits modification.

2.2.2 Creating Implementations

Once the du exists, a user may begin to enter implementations. As part of *implementation creation*, the user supplies the source module. The Library checks to ensure that the interface found in the source code matches the abstraction's interface as defined by the du's interface specification.

Sometimes users want to create several binary components using the same source module. For example, an implementation compiled twice with different options or for different target machines has multiple binary versions. In the Library, an implementation can have only one binary module. To associate several binaries with the same source, the user creates multiple implementations, each having a distinct binary module but sharing the same source module.

The other two parts of an implementation are produced by compiling the source module. The binary module is the compiled machine code version together with information needed by the linker to resolve external references. The abstraction bindings record the dus representing subsidiary abstractions referred to by the implementation.

To determine the bindings of external references and to support type-checking, the compiler uses a *compilation environment*, or *CE*, supplied by the user. The CE contains mappings between abstraction names and the dus representing them. When the compiler encounters an external reference, it looks in the CE for the du bound to the abstraction name. It records the binding in the implementation's abstraction bindings, and uses the du's interface specification to verify type correctness.

Use of a separate compilation environment allows users to reuse abstraction names. The names of dus are unique across the Library: no two dus may have the same name. If the abstraction name in an implementation's source module also specifies the du representing it, then users have to be aware of all the dus in the Library to avoid conflict when naming a new abstraction. Use of the CE circumvents this problem by decoupling abstraction and du names.

Note that this approach to compilation binds an implementation to its subsidiary abstractions at compile-time. Because type-checking examines the du representing a subsidiary abstraction, type errors would result if an different du were used to represent the abstraction after compilation. Consequently, substituting a different du is not allowed.

The components of an implementation are controlled by the Library to protect against modification. The source module is supplied by the user as part of implementation creation and may never be changed. The binary module and bindings are produced by compilation; an implementation may be compiled only once. To achieve the effects of source module modification or implementation recompilation, the user must create a new implementation.

2.2.3 Building Compositions

Once implementations have been created and compiled for each of the abstractions in the program, the user can bind them together to make an executable program. To create a new composition, the user supplies the set of implementations which form its components. The linker produces the executable module by extracting the binary modules from these implementations, resolving their external references, and linking them together.

It is the user's responsibility to identify the component implementations. To successfully create an executable form of a program, the set of implementations should satisfy two conditions. First, the set of component implementations must contain an implementation of the program's top-level abstraction. Second, the set must also contain an implementation for every abstraction used by any implementation in the set. If one of the implementations in the set refers to an abstraction for which there is no implementation, the linker will find an unresolvable external reference and will be unable to build the executable module.

The user can identify a satisfactory set of implementations by first selecting an implementation for the program's top-level abstraction, then selecting an implementation for each of its subsidiary abstractions, then for their subsidiary abstractions, and so on. Chapter 6 contains a proposal for building a composition using this algorithm. In general, a set of component implementations meets the two conditions if it contains every implementation on some dependency graph that has a node with no ancestors implementing the top-level abstraction of the program.

In this thesis we assume that the set of component implementations contains exactly one implementation per abstraction and that all uses of an abstraction in a composition invoke the same implementation. However, sometimes it can be useful to include multiple implementations of an abstraction in the same program. For example, one program might contain two uses of a sorting abstraction. If one use sorts only a small number of items, and the other sorts a large number, then two different sorting implementations may be preferred.

Although it is always safe to include two implementations of a procedural or control abstraction in the same program, a program containing multiple implementations of a data abstraction may yield erroneous results. An implementation of a data abstraction specifies a data representation as well as a set of operations. The operations provided by an implementation have certain expectations about objects of the data type, such as what representation is used. Two implementations may have entirely different expectations. For example, an abstraction representing points on a graph could have two implementations: one assuming polar coordinates and the other assuming Cartesian. If multiple implementations of a data abstraction exist in the same program, then an object conforming to one implementation's expectations may be manipulated by operations having incompatible expectations.

For this thesis, we make the simplifying assumption that multiple implementations of an abstraction are not permitted. This restriction is enforced by the flat set structure used in the

Library to represent the set of component implementations. If greater flexibility is desired, the set structure can be replaced with a more complex graph structure, parallelling a composition's dependency graph, that permits distinct uses of an abstraction to invoke different implementations.

The two components of a composition are controlled by the Library to protect against modification. The user is permitted to modify the set of component implementations until the linker is invoked on the composition. After the executable module has been created, no changes are permitted to the composition. To achieve the effect of changes, the user must create a new composition.

2.3 A Model of Abstraction Evolution

This section presents our view of how an abstraction evolves and new implementations are developed. Our model of development recognizes two important relationships among the implementations of an abstraction. One relationship partitions the set of implementations into classes of similar implementations, while the other corresponds to a partial order based on the notion of improvement.

We use this model for two purposes in this thesis. First, it provides a basis for understanding the set of implementations in a du. It imposes order on the set by exposing the historical relationships between individual implementations. Second, the model serves as the foundation of software development conventions used in later chapters of this work. Note that our work does not require the use of this particular model of abstraction evolutionl; it could be adapted to use other models and conventions.

2.3.1 Alternatives

Often the set of implementations in a du can be divided into several distinct subsets, each of which represents an alternative way to implement the abstraction. For example, a sorting abstraction might be partitioned into subsets based on the choice of algorithm. Some implementations might use a bubblesort algorithm, others insertion sort, and still others quicksort. Typically a user develops a series of implementations in a single subset rather than switching from subset to subset. A user of the sorting abstraction might develop a number of bubblesort implementations, then deciding it is too inefficient, abandon bubblesort to create a series of quicksort implementations. It would be unusual for him to alternate between implementations of bubblesort and quicksort.

We call this partitioning phenomenon the alternatives relationship among the implementations of an abstractions. We say the subsets are the *implementation alternatives of a du*, or *alternatives* for short. Partitioning a du into alternatives enhances a user's understanding of a du by exposing the significant differences between implementations. Implementations belonging to distinct alternatives are inherently different; those in a single alternative are subject only to more minor variations and are in some sense substitutable. Each alternative meets a particular set of needs and is appropriate for different conditions. No alternative is better than another for all circumstances.

Many criteria can be used to distinguish alternatives. As in the sorting abstraction, algorithm often forms a useful basis for partitioning many dus, but requirements and performance properties may dictate segregation in others. The users of a du determine what differences are significant.

2.3.2 Supersession

As an abstraction evolves over time, new implementations are added to correct bugs, satisfy changing requirements, and so on. Where the alternatives mechanism partitions a du's set of implementations into coequal subsets, the supersession relation captures the notion of improvement between implementations. An implementation supersedes another when the former improves in some sense upon the latter, for example by correcting an error or by eliminating inefficiency. What constitutes improvement can only be determined by a user familiar with the set of implementations. Because improvement is a transitive quality, supersession is also transitive.

Users implicitly rely on an understanding of supersession relations to select among the implementations in a du. Most file systems and software databases use historical derivation as an approximation of supersession. The implementations of an abstraction are ordered linearly by creation date, with newer implementations assumed to supersede older implementations. Unfortunately, creation order and supersession are not equivalent. When two implementations are tuned to meet different requirements, neither can be said to supersede the other. Similarly, a debugging version is different but not necessarily better than the implementation from with it was created. Creation order is an inadequate model of supersession because its linear view does not reflect the non-linear nature of abstraction evolution.

Because understanding the supersession relations among implementations is critical to understanding the evolution of an abstraction, the diagrams in this thesis display the set of implementations in a du as a directed graph who nodes represent implementations and whose arcs represent supersession relations. The implementation at the head of the arc supersedes (and thus is considered a better implementation) than the implementation at the tail. For example, Figure 2-2 depicts a du containing four implementations represented by numbered circles. The numbers order the set of implementations by creation order. The directed arcs denoting supersession relations show that implementation 4 supersedes implementation 2 which in turn supersedes implementation 1. Implementation 1 is also superseded by implementation 3.

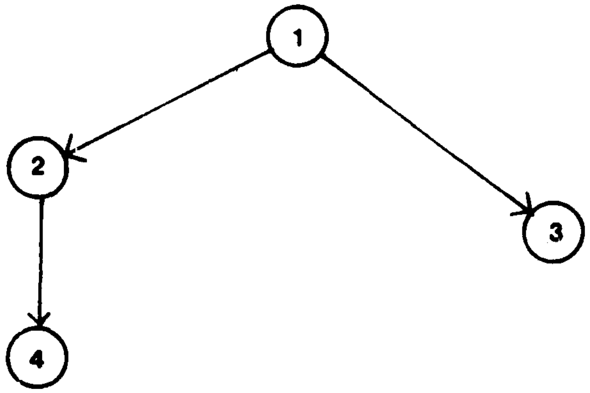


Figure 2-2:A DU With Supersession Relationships

2.4 Some Notation

In this section we introduce some notation used in examples later in the text.

We use CLU and CLU syntax as the basis for much of our work. In CLU, an invocation of a procedure named P has the form

P (arg1, arg2,...)

where arg1, arg2, and so on are P's arguments. P can return any number of results, depending on its interface type. An invocation of an operation named O provided by a data abstraction T has the form

T$O (arg1, arg2,...).

Again, arg1, arg2 and so on are O's arguments, and O may return results, depending on its interface. In the examples in this thesis, all invocations return a single result object.

CLU provides the ability to write invocations more conveniently through the use of syntactic sugars. Syntactic sugars are textual substitutions: the compiler recognizes a textual pattern and substitutes an invocation of a predetermined form. For example, one sugar replaces the pattern

<arg1> <comparison operator> <arg2>

with an invocation of the form

T$<corresponding comparison operation> (<arg1>, <arg2>)

when the arguments are of type T. Thus, the program text

T$Equal (A, B)

is substituted for

A = B

if A and B are of type T. In this thesis, we use only sugars with obvious meanings, such as sugars invoking standard comparison operations or boolean operations.

Finally, although in this thesis we do not address naming of components of a du other than individual implementations, we frequently need to refer to the whole set of implementations contained in a du. We use the notation

$$\mathcal{I}$$

to denote the set of implementations in a du.

Chapter Three

Analysis of Naming Requirements

Multiple implementations of an abstraction share a common specification but may exhibit differing behavior for properties not addressed by the specification. To identify the implementation that best suits his needs, a user must understand these differences. Without help, it is difficult for the user to comprehend the large volume of information describing an abstraction's implementations. Expressive names can reduce the burden on the user by recording the properties of implementations. Such names support implementation identification by documenting the significant differences between implementations and by providing a medium through which users can share their insights.

Users create a new name for an implementation whenever they identify an important aspect of an implementation's behavior. In this way a description of each implementation's behavior is associated directly with the implementation. When a user selects an implementation for use, he describes the behavior he wants. This description can be matched against the description of each implementation to find the most satisfactory choice.

In this chapter, we analyze the problem of implementation identification to determine what characteristics of a naming scheme support the above scenario. As a basis for our analysis, we trace the development of an abstraction in the context of the CLU Library and programming support environment. Our example is a hashing abstraction shared by two projects. Then, examining the ways in which its implementations can be identified, we identify six characteristics of naming systems that support implementation identification.

3.1 Development of a Hashing Abstraction: An Example

A programmer from Project X initiates development of the hashing abstraction, specifying its interface and creating a du for it in the Library. At this point, he can begin to enter implementations for it. His first implementation uses a small dense hash table, whose entries point to linked lists of buckets stored separately from the hash table in an external storage area. The expected loadfactor of the table is high, so each bucket is made large enough to accommodate several entries. Several implementations are needed to correct errors, but the fourth attempt produces a working version.

Before starting development, a programmer from Project Y looks in the Library for existing abstractions that can be shared. He discovers the hashing abstraction du created by Project X and determines that its specification meets his needs. Inspecting the set of implementations, however, he finds none that satisfy his requirements. He creates his own, entering them in the du now shared by the two projects.

His first implementation uses open addressing with a linear probe sequence. A second version, needed to fix a bug, is incorporated into the Project Y's system. Experience with the application, however, shows that the linear probe sequence is a bottleneck. The programmer responds with a new version using a quadratic probe sequence.

The three open addressing implementations contain the code for the probe sequence embedded within them. In a fourth open addressing implementation, the programmer abstracts away the probe sequence, invoking it as a subsidiary abstraction. He also enters a du for the new probe sequence abstraction into the Library. Project Y successfully incorporates this new open addressing version into its system.

In the meantime, Project X's programmer proposes an alternate buckets version implementation 7's is quadratic. Although implementation 7 was created by editing implementation 6, it is not an improvement; implementations 6 and 7 are really variations, each of which might be useful under different conditions. Thus, no arc connects 6 and 7. Both implementations 6 and 7 improve on 5, however, so arcs connect the latter with each of the former. Implementation 8 is the open addressing scheme with the probe sequence decoupled as a separate abstraction and is therefore an improvement on both 6 and 7. Implementations 9-11 use external single-valued buckets. As with implementations 6 and 7, implementation 9 is a variation rather than an improvement on imnlementation 4 so no arc joins them. Implementations 10 and 11 share the same CLU source but were compiled using different options. Thus, they both improve on implementation 9, but neither improves on the other.

The snapshot view of Figure 3-1 offers the user a basic understanding of the state of the hashing abstraction at a particular instant in time. The set of implementations can be partitioned into two subsets which represent two alternative implementation schemes: those implementations using open addressing and those using external buckets. Every implementation developed so far is a member of one of these alternatives. The buckets alternative can be further subdivided into those with single-valued buckets and those with larger multi-valued buckets. The directed arcs trace the debugging history. Branches or separate subtrees in the history graph mark alternatives, such as the choice of probe sequence in the open addressing subset. The fully debugged and usable implementations are found at the leaves of the graph.

3.2 Identifying Implementations of the Hashing Abstraction

A user selects an implementation of an abstraction based on its behavior and properties. Which aspects of an implementation are considered significant depends both on the user and on the circumstances. Two users may choose to use widely differing qualities and even a single user may consider distinct properties in different circumstances. In this section, we examine what qualities influence users of the hashing abstraction in selecting an implementation.

At first, while developing the initial implementations, Project X's programmer almost always chooses to work with the most recently created version. He relies primarily on his understanding of the debugging history. Later, he also considers external factors that influence the implementations, such as performance requirements imposed on the abstraction by the Project. If he returns to the abstraction after a long absence, he may be interested in more basic properties of the implementations, such as the major differences in algorithm.

Other members of Project X are likely to use status information and configuration management information: they need to know which implementations have been approved for use by the rest of the project and whether or not the approved versions contain any bugs.

If Project X has a long lifetime, the hashing abstraction may be turned over to a new programmer for maintenance. The maintainer is concerned with some form of absolute identity and with derivation information. If the hashing routine in Project X's production version breaks, he needs to know exactly which implementation is involved and how to recreate the faulty system.

On discovering the du in the library, the Project Y programmer responsible for the hashing abstraction is interested in many different aspects of each of the existing implementations because he has no knowledge whatsoever of the abstraction. He needs to understand the implementations' debugging histories; he needs to know which implementations have been tested and which contain bugs; he needs to know if there are any authorization restrictions that would prevent him from sharing any of the implementations or if there is a default implementation recommended for all users. Once the Project Y programmer is familiar with the du and begins to enter new implementations himself, he mirrors Project X's programmer, primarily considering debugging history and external requirements like hash table loadfactor.

Other Project Y members rely on configuration management and performance information. If an error is found, they may also be interested in knowing that the optimized version incorporated into the Project Y system shares its CLU source with an unoptimized implementation.

3.3 Naming Support for Implementation Identification

Using the hashing abstraction as a starting point, we can identify six characteristics of naming systems that support implementation identification.

3.3.1 Expressiveness

Names must have the capacity to convey a diverse collection of information about implementations in a form meaningful to people. Names supporting implementation identification act as a kind of documentation, recording properties of the implementations they are associated with. Human convenience, rather than machine efficiency, is the goal.

The naming system should allow a user to express a wide range of information, corresponding to the many different properties he considers helpful in discriminating among implementations. In the hashing example, we found that the two programmers relied on debugging history, external requirements, implementation details (like algorithm), correctness, and so on, while other project members were concerned with information about status, configuration management, and implementation derivation. In general, the information expressed as names can range from very concrete data like implementation storage requirements (perhaps derived automatically by the programming environment) to high-level descriptions (provided by the user) of semantic properties of implementations.

It is not possible to delineate the set of information relevant to implementation identification. The user determines what information should be conveyed. An item of information is relevant if it helps to distinguish between several implementations or otherwise aids a user in understanding the set of implementations contained in a du. Each user of an implementation has his own requirements, needs, and tastes, and thus may desire different information. In addition, much useful information is abstraction-specific, for example, the choice of algorithm or data structures. The naming scheme cannot predetermine the classes of useful information.

Although it is impossible to enumerate the particular items that are important, generalizing from the hashing example leads to a number of classes of information that will often be useful in distinguishing among the implementations found in a du. The following list is not intended to be complete. It merely suggests the range of information that can be useful in solving the problem of version identification.

- **Identification, derivation and historical information.** This class of information provides some form of absolute identification for each implementation, perhaps in the form of a unique identifier. It also specifies how each implementation was created and what role it plays in the evolution of the abstraction. Such information includes the date the implementation was created, the name of the user who created it, the version and parameters of any tools used to create it, a comment explaining why it was created and a list of the subsidiary abstractions used. It should specify the implementation's role in the debugging history. It might also describe the source of ideas used in this implementation, for example, when an implementation is ported from another installation or is based on a published algorithm.

- **Role in compositions.** This information at a minimum specifies what compositions contain each implementation. It could also specify a default implementation to be used by the general public, and configuration management information for specific projects. For example, it might name the implementations released to the public or to specific user groups, or it might identify a particular project or subsystem to which an implementation belongs. It could also describe an implementation's use to date, detailing how long it has been in service and providing statistics.

- **External requirements.** External factors such as project conventions, target hardware, and software performance needs may influence the functional characteristics of some implementations. For example, external requirements could specify performance or optimization levels, target hardware, programming language conventions or minor customizations. This information might also document instrumentation, assertion generation, conditional compilation, or host/target dependencies.

- **Algorithmic or Other Descriptive Properties.** This class of information describes basic properties of an implementation. If the specification and external requirements define the externally visible behavior of a black box, then this information provides a view inside. It describes how an implementation works, what algorithm is followed, and what data are used. It might include an invariant, an abstraction function or a set of relevant assertions. It could offer advice on how to maintain the implementation or run it optimally.

- **Status.** This information describes the status of individual implementations. It should include testing history and results, a description of known bugs and perhaps advice on avoiding or correcting them. It could also include a performance evaluation, describing how well an implementation meets its specification and requirements. It might specify what phase of the software lifecycle an implementation represents (e.g., experimental versus production).

- **Authorization.** Authorization information controls access to implementations in the du.

3.3.2 Multiplicity

A second property of naming systems supporting implementation identification is multiplicity of names. Many distinct aspects of an implementation may be of interest to users in different situations. A single implementation may at various times be identified by its derivation history, its functional or performance properties, its status, or its role in compositions. For example, the Project X programmer responsible for the hashing abstraction used information about debugging history, external requirements satisfied, and implementation details to identify implementations, but other members of the project used status and configuration management information.

Consequently, each implementation may need more than one name. A naming scheme limited to a single name per implementation cannot express the full range of information required by several programmers engaged in diverse activities.

3.3.3 Non-uniqueness

A name may not identify an implementation uniquely; instead it may be shared among several implementations. Names express information about properties of the implementations they are associated with. Two implementations, however, may exhibit the same behavior with respect to a particular property. A name reflecting that property will be identical for both implementations.

For example, a name in the hashing abstraction might describe whether the implementation used open addressing or external buckets. Since implementations 5 through 8 all use open addressing, behaving identically with respect to this property, they would share a common name.

A consequence of non-uniqueness is that names are ambiguous. Multiple names may be required to unambiguously identify a particular implementation.

3.3.4 Independence

The naming scheme should offer users the ability to create a new name for an implementation independently, without considering other implementations or their names. When a user determines that a property is a significant factor in understanding an implementation's behavior, he creates a new name for the implementation reflecting that property. He should not be required to supply corresponding names for other implementations, documenting their behavior with respect to that property. Such a requirement would place an unreasonable burden on the user.

An additional argument in favor of name independence is that the property described by a new name may not be pertinent to other implementations. Some names do reflect properties applicable to the implementations of all abstractions. For example, all implementations have a creator and a derivation; other qualities are shared as well. Some qualities, however, are specific to a particular abstraction. For example, there may be several possible algorithms for a particular abstraction. Although the concept "algorithm" is applicable to all abstractions, the particular choices are specific to the abstraction. Even within a du, the significant aspects of one implementation may be either irrelevant or uninteresting with respect to another. In the hashing example, information about the probe sequence is relevant only to those implementations using open addressing. In contrast, information about hash table size applies to all implementations.

Multiple users sharing a du may also cause information use to be partitioned among subsets of the implementations. If each of the sharers is interested in a distinct subset, collectively they superimpose their differing information needs on the du. In the hashing example. Project Y's information needs dominate the open addressing implementations because Project X is not interested in those implementations.

Note that it is sometimes desirable to recognize dependencies among the names of an implementation. Names expressing related properties may be more informative viewed together. A name documenting bugs in an implementation complements a name recording its test status. Dependencies may also exist among the names of distinct implementations. Names recording the properties of one implementation relative to another introduce a question of consistency. Consequently, a user creating a new name for an implementation may choose to consider the already existing names of implementations. However, other names do not share in dependency relationships that contribute significantly to a user's understanding of the associated implementations. Users should have the ability to treat these names independently.

3.3.5 Commonality

Although the naming system should not require the user to record corresponding information about multiple implementations, it is frequently useful to compare the behavior of two implementations with respect to the same quality. When both implementations are identified by names reflecting one property, the naming scheme should expose this commonality. Pairs of implementations with identical behavior should have identical names; pairs with differing behavior should have distinct but related names, highlighting the fact that they express the same class of information.

In the hashing abstraction example, the four open addressing implementations could all have names identifying the probe sequence used. The two using a linear probe sequence would have identical names. The other two, using a quadratic probe sequence and a probe sequence decoupled from the hashing abstraction, would have different names. It should be recognizable, however, that all four names express information about the same property.

3.3.6 Mutability

The collection of names identifying an implementation may change over time. It is a mutable set, growing as different aspects of the implementation become of interest. A user may create new names at his own discretion.

A user may also modify names. Some properties of an implementation are fixed over time. Others, however, change as the abstraction evolves or as a user's perception of the abstraction changes. For example, names reflecting status information may need to be modified as an implementation is tested. Similarly, names reporting the properties of one implementation relative to others may change as new implementations are developed.

3.4 Summary

In this chapter, we used an example tracing the development of a hashing abstraction to investigate the naming of implementations. Our analysis found several characteristics of naming systems support users in identifying implementations. First, names must be expressive, recording the properties of implementations in a form meaningful to people. Names may be used to express a wide range of information, chosen at the user's discretion. Second, because many different aspects of an implementation may be important in different situations, each implementation may be identified by several names. Each name may also be associated with multiple implementations: implementations exhibiting the same behavior with respect to some property may share an identical name reflecting that property. Consequently, multiple names may be required to identify an implementation uniquely. Another characteristic of naming systems that supports implementation identification is name independence. Although dependencies may exist among the information expressed by some names, the user may often want to view names independently. The naming scheme should offer him the ability to do so. Commonality of information, however, should be exposed. In particular, two names conveying information about the same property should be related. A final characteristic supporting implementation identification is mutability. Over time, a user's view of the significant features of an implementation may change. The naming system must allow him to create and modify names as needed. As a result, the set of names identifying an implementation may evolve.

Chapter Four

Naming Implementations in the CLU Library

In this chapter we propose a naming system for use in the CLU programming support environment. We discuss the structure and use of names identifying the implementations found in description units in the Library. We also present operations for creating and inspecting names. Additional operations allowing users to retrieve implementations from the Library are described in Chapters 5 and 6. Our proposal exhibits the six characteristics supporting implementation identification enumerated by Chapter 3.

The chapter is organized as follows. Section 4.1 introduces the abstract concepts that form the basis of our proposal. They are abstract in the sense that they are neither programming language nor programming support environment dependent. Section 4.2 adapts these concepts for use in the CLU programming support environment, and develops a proposal for naming implementations in the CLU Library. Section 4.3 illustrate our proposal with examples of names and demonstrates use of the name manipulation operations. Section 4.4 assesses the impact on our naming system of a problem occurring in all long-lived systems written in CLU, such as the Library. Finally, our proposal provides for some standard classes of names. Section 4.5 suggests some sample classes for use in the CLU programming support environment.

4.1 Basics Concepts

In this section we introduce basic concepts that are the foundation of our proposal for naming implementations: names, classes of names, and operations for manipulating names.

A name in our proposal takes the form of an attribute that is associated in the Library with the implementation being named. An attribute records some fact about the implementation that the user creating the name considers significant. We introduce the notion of a class of names, represented by a class of attributes, to accommodate commonality among multiple names documenting the same property of implementations. Each class corresponds to a property of implementations. Individual attributes describe a particular implementation's behavior with respect to the property. We also present abstract definitions of operations for manipulating names (attributes) and classes of names (attribute classes).

After presenting the foundations of our proposal, we argue that these basic concepts can be combined in a naming system that provides all six of the characteristics enumerated in Chapter 3 as supporting implementation identification.

4.1.1 Attributes

An attribute is a pair, consisting of an attribute name and a value. The attribute name reflects a class of information, expressing the intended meaning of the attribute and providing a context in which to interpret the attribute's value. For example, a user of the hashing abstraction of Chapter 3 could name an implementation with an attribute describing its hashtable size. The attribute name informs users that the value represents the implementation's table size. The integer value records the specific table size used by the implementation.

The naming scheme is used as follows. A user creates a new name for an implementation to document some property of its behavior. To do so, he invokes an attribute creation operation of the programming support environment, specifying the implementation, an attribute name representing the property, and a value corresponding to the implementation's behavior with respect to the property. The operation associates in the programming support environment a mapping between the attribute name and value with the implementation. If the implementation's behavior with respect to property changes later, the user can modify the name using the same operation to replace the attribute's value with a new value corresponding to the modified behavior.

A user can also inspect the names of an implementation or a set of implementations. Using one programming support environment operation, a user can inspect all the names of an implementation. The operation retrieves from the environment copies of all the attributes describing an implementation specified by the user. Another operation reports the behavior of an implementation with respect to a given property. When the user specifies an implementation and an attribute name representing the property, the operation retrieves the value of any attribute associated with the implementation that has the attribute name specified by the user. A third inspection operation allows a user to inspect all the names on a set of implementations that document a single property. The user specifies a set of implementations and an attribute name. The operation locates all the attributes having the specified attribute name associated with any of the implementations in the set.

Names in the form of attributes can also be used to retrieve implementations from the programming support environment. Implementation retrieval, however, is a more complex operation. We defer its discussion to Chapters 5 and 6, each of which proposes an operation using attributes to retrieve implementations from a programming support environment.

4.1.2 Attribute Classes

Each attribute is a member of a class of attributes, all sharing the ˍame attribute name. An attribute class corresponds to a property of implementations; its member attributes record the behavior of specific implementations with respect to that property. For example, the hashtable attribute described above is a member of a class of attributes whose attribute name is shared by all members of the class. Each attribute in the class reports the hashtable size used by the associated implementation.

Users may often want to compare two attributes in the same class. Consequently, we require all attributes in a class to draw their values from the same set, which is called the domain of the class. The domain defines the set of legal values for members of the class. In the hashtable example, the domain associated with the class is the set of positive integers: all attributes in the class must have positive integral values. A programming environment can enforce the requirement that every attribute draw its value from its class's domain with a domain checking mechanism: before allowing an attribute to be created or modified, the programming environment examines the attribute value to verify that the value is contained in the attribute's domain.

A programming support environment provides a set of standard attribute classes representing basic properties applicable to all implementations. These may include derivation, algorithm, status, properties relating to program development conventions in use, and so on. The attribute names and corresponding domains of the standard attribute classes are predetermined and built into the programming support environment.

It is not possible to enumerate a set of standard classes suitable for all programming support environments because many useful properties may be dependent on the programming language, environment, and conventions in use. Section 4.5 suggests sample standard attribute classes that might be useful in the CLU programming support environment. Experience with a naming system of this sort is needed to determine precisely which standard classes should be provided.

The set of standard attribute classes will not satisfy the needs of all users because they only record properties common to all implementations of all abstractions. However, users may also want to record properties of implementations that are specific to an abstraction or to a project. Our proposal allows users to create new attribute classes to express these abstraction or project specific properties. New attribute classes created by users are called customized attribute classes.

We view the standard attribute classes as the basic naming structure for implementations in a programming support environment. Standard classes provide a common base of properties applicable to all implementations and understood by all users. Customized attribute classes offer flexibility, allowing a user to extend this basic naming structure. Users should be able to rely on the standard classes for the majority of their needs. Only users with special needs should require this added flexibility. If the set of standard classes provided by a programming support environment encompasses a reasonably complete range of properties of implementations, then users should rarely need to create new attribute classes.

An attribute class is defined by binding a domain to an attribute name. For standard attribute classes, the definition is built into the programming support environment. For customized classes, however, the user creating the new class must provide a definition. An attribute class definition operation provided by the programming environment allows a user to define a new customized class by specifying the domain of legal values to be bound to the new attribute name.

The class definition operation also requires the user creating a customized class to provide a description of the new class's intended meaning. This description promotes a consensus among users about the use of a customized attribute class. A user encountering an attribute from a customized attribute class for the first time can look up its meaning in the attribute's class definition.

4.1.3 Abstract Operations

In this section we propose five operations for creating and inspecting attributes. Only abstract syntax is presented; the choice of human interface will depend on the philosophy and conventions of the programming support environment.

The operation DefineAttribute defines a new class of names. It has the form

DefineAttribute <AttributeName> <Domain> <Description>.

DefineAttribute associates the set of values <Domain> with the name <AttributeNames> to define a new class of attributes. The <Description> is user-provided text documenting the intended meaning of the attribute class and its member attributes.

The operation SetAttribute creates a new name or modifies an existing one. It has the form

SetAttribute <Implementation> <AttributeName> <Value>

where <Value> is an expression denoting a member of the domain associated with the name <AttributeName> in some attribute class definition. For an implementation <Implementation> with no already existing attribute in the class <AttributeName>, this operation creates a new attribute with attribute name <AttributeName> and value <Value>. If <Implementation> already has an attribute member of the class, the attribute's value is modified to be <Value>. This operation fails if no class with attribute name <AttributeName> has been defined.

The operation ReadAttribute allows a user to inspect a single name from a class of names. It has the form

ReadAttribute <Implementation> <AttributeName>.

This operation returns the value of the attribute with attribute name <AttributeName> associated with implementation <Implementation>. It fails if <Implementation> does not have an attribute member of class <AttributeName>.

The operation ReadAllAttributes allows a user to inspect all the names of an implementation. It has the form

ReadAllAttributes <Implementation>.

This operation returns a set of mappings from attribute names to attribute values, where each mapping corresponds to an attribute associated with implementation <Implementation>.

The operation ReadAttributeOnSet allows a user to inspect all members of a class of names associated with any implementation in a set of implementations. It has the form

ReadAttributeOnSet <Set[Implementations]> <AttributeName>.

This operations collects the values of all instances of an attribute class across a set of implementations. It returns a set of mappings from implementations to attribute values. For each of the implementations in <Set[Implementations]> having an attribute in the class <AttributeName>, a mapping from the implementation to the attribute's value is returned.

4.1.4 Discussion

Combining the concepts described above leads to a naming system that exhibits the six characteristics enumerated in Chapter 3 supporting implementation identification. First, it allows the user to convey a wide range of information as names. Users can define attribute classes to represent any property of an implementation they choose. String attribute names express the intended meaning of the attribute, providing a context in which to interpret the attribute's value. The value documents the behavior of an implementation with respect to the property represented by the attribute.

Attributes provide name multiplicity, non-uniqueness, and independence. With the SetAttribute operation the user explicitly identifies the attribute name and value comprising a new attribute and the implementation to which the attribute will be bound. No constraints on this operation prevent the user from applying more than one attribute to the same implementation, or the same attribute to multiple implementations. The SetAttribute operation allows the user to create a name for an individual implementation, without reference to other implementations or their names.

Commonality among names is expressed via the attribute class mechanism. Two attributes belonging to the same attribute class reflect the same property of implementations and convey the same kind of information. Because all the attributes in a class draw their values from the same set, the class's domain, the behavior of two implementations with respect to the same property can be compared. The shared attribute name highlights commonality: users can easily recognize that two attributes reflect the same property of implementations because the attributes share a common attribute name.

Finally, the naming scheme allows users to create and modify names at will. The SetAttribute operation can be invoked an any time to create an attribute for an implementation or to modify an attribute's value. Thus, the set of names identifying an implementation may change over time.

4.2 A Naming System for the CLU Library

The preceding section introduced a basic naming structure applicable in any programming support environment. For the rest of the chapter we focus on the CLU programming support environment. In this section we examine the issues that arise when attempting to embed this general naming structure in the CLU Library. Such issues include the nature of attribute names. the representation of attribute values and attribute class domains. and the use of standard attribute classes. We suggest ways to adapt the basic concepts presented above for use in the CLU programming support environment and develop a complete proposal for using attributes to name implementations in the Library.

4.2.1 Attribute Names

In this section we consider issues relating to the specification of attribute names for use in the CLU Library.

The CLU Library is a centralized storage facility used by all CLU users. Because the Library is shared by many users, contention over attribute names may be a problem. Two users may want to convey different information with attributes having the same attribute name. For example, two projects may want to define an attribute class to record information about implementations promoted as official releases. If one project divides releases into releases to the customer (often called baselines) and minor releases to project members (called updates) while the other has only one kind of release, the attribute representing release number will be a point of conflict between the two projects. The first project requires a composite value to name both baseline and update while the second needs only a simple value, omitting the update number.

Globally unique attribute names force all users to agree on common meanings and definitions. This approach is unacceptable because it does not allow users to tailor attributes to their own needs without referring to all other users.

To support reuse of attribute names, we propose a context mechanism. Users define attribute classes within a context. Attribute names are qualified by the name of the context in which their class definition can be found. Consequently, users with different contexts can share an *unqualified attribute name* without conflict. Two attributes with different qualified attribute names but the same unqualified attribute name may in fact be associated with the same implementation. Contexts provides a finer grain of name control by allowing attribute classes to be defined locally. Using separate contexts, users can reuse names. Controlled sharing of attribute classes among project members or other interested individuals can be accomplished by sharing contexts.

We expect each context to be organized around a concept, such as a project or an abstraction, that unifies the contained class definitions. In particular, each du has a du-specific context (perhaps empty) containing the definitions of classes applicable only to itself. For example, the hashing abstraction's du-specific context could include definitions for attribute classes recording probe sequence and hashtable size. Independent contexts, containing the definitions of classes applicable across several dus, represent concepts such projects, subsystems, or a particular programmer's work.

Although all operations for manipulating attributes in the Library require the use of qualified attribute names, we provide a mechanism supporting the use of unqualified attribute names in some circumstances. An unqualified attribute reference is permitted when the Library can deduce the name of the context containing the attribute's class definition. The definitions for standard attributes classes are contained in a special context built into the

Library. Unqualified attribute names may always be used for attributes from the standard attribute classes. Users may refer to customized attribute classes with unqualified names by first invoking a Library operation to specify a default context. The Library automatically uses the default context to resolve all unqualified references to customized attributes. The default may be overridden by explicitly qualifying an attribute reference.

We use unqualified attribute references for the examples in this thesis. The existence of a default context containing definitions for all our customized attribute classes is assumed.

4.2.2 Attribute Values and Attribute Class Domains

In this section we consider issues concerning representation of attribute values and attribute class domains in the CLU programming environment.

In the CLU programming support environment, we represent attribute values as CLU objects, and attribute class domains as the set of objects created using a single composition of a data abstraction. We call this composition the *domain specifier* of the attribute class and call the abstraction implemented by the composition the *domain type* of the attribute class and its member attributes.

Our rationale for representing attribute values and class domains in this way is as follows. In CLU, all values are represented by typed objects. Consequently, an attribute value is a single typed object and an attribute is a mapping from an attribute name to an object. A domain corresponds to a set of objects. The CLU data abstraction mechanism, defining a set of objects and some operations for manipulating them, is a natural parallel for this concept. A domain's set of values corresponds to the abstraction's set of objects. An individual attribute's value is represented by one of the abstraction's objects. The set of values defined by a data abstraction thus provides an intuitive representation for an attribute class domain.

In CLU, an object may be manipulated only using the operations provided by its type. In addition, it may only be manipulated using the particular composition of its type used to create the object originally. This is a consequence of the constraint discussed in Section 2.2.3 regarding multiple instances of a data abstraction. Each implementation of a data abstraction has certain expectations about the representation and invariant properties of objects of the type. A computation in which an operation provided by one implementation is used to manipulate an object created using another implementation may yield erroneous results because the two implementations may have incompatible expectations. We therefore disallow mixing two compositions of a data abstractions in a single computation.

In the course of creating and maintaining attributes, the Library must manipulate the objects representing attribute values. For example, creating or modifying an attribute may require creating an new object to represent its value. Similarly, when an attribute's value is inspected, the Library makes a copy of the value object. Consequently, the Library must have access to every composition used to create an object stored in the Library as an attribute value.

Although an attribute class domain, as a set of values, maps intuitively into the set of values defined by a data abstraction, we prefer to further constrain the representation of domains. Using a composition of a data abstraction to generate the set of objects from which an attribute class's values may be chosen guarantees that a single composition may safely be used by the Library to manipulate all objects representing the values of attributes in the class. In contrast, no single composition may be used when a domain is identified only with a data abstraction.

In addition, using a composition of a data abstraction to specify a domain supports the comparison of attribute values. It is not possible to use an operation provided by a data type to compare two objects created with distinct compositions of the type. For example, suppose two objects x_1 and x_2 created using different compositions C_1 and C_2 of type T represent the values of two attributes in the same class. No operation provided by type T can compare the values of these attributes, because x_1 can only be manipulated using composition C_1 while x_2 can only be touched with C_2. Requiring all objects representing values in an attribute class to be created using the same composition allows the Library to invoke the comparison operators provided by the domain type to compare any two attributes in the class.

4.2.2.1 Domain Types

An important issue concerns the range of types that may be specified as domain types. One restriction is that each domain type is required to provide a specific set of operations needed by the Library to manipulate attribute value objects. The exact set of operations required by the Library is not defined by this thesis. We anticipate, however, that it will include operators to create new objects, to copy objects, to compare two objects for equality, and so on. Abstractions that do not provide the required operations may not be used as domain types.

Given this restriction, what abstractions may be selected as domain types? For example, are user-defined abstractions acceptable? The CLU programming support environment allows any type providing the required operations, including user-defined types, to serve as a domain type. Section 4.2.2.2 discusses an alternate proposal restricting domain types to a predetermined set of abstractions.

Our approach does not restrict domain types to a predetermined set. Users may choose arbitrary types, including user-defined types, to represent domains. This approach allows users to represent values in an intuitive manner. With user-defined data abstractions, users can model a domain precisely, creating a new data abstraction whose objects correspond one-for-one with the domain's values.

A consequence of this is that domain checking is equivalent to checking the composition of an attribute value object. Whenever a user creates or modifies an attribute, the programming support environment must perform a domain check to ensure that the new attribute value is an element of the attribute's domain. In the Library, the domain check is accomplished by verifying that the object representing the new value was created by the attribute's domain specifier, the composition generating all legal values for the attribute class. Because the set of objects created with this composition represents exactly those values contained in the attribute's domain, verifying that the attribute value object was created with the domain specifier also ensures that the value falls within the attribute's domain.

There are several disadvantages to allowing arbitrary types to be used as domain types. The first concerns the safety of the information represented in the Library as attributes. When a user defines a new attribute class, he provides a composition to be used as the domain specifier. The Library uses this composition to create and manipulate objects representing attribute values. If the domain specifier implements a user-defined type, then the Library invokes user-supplied code to maintain the values of attributes in the class.

The Library assumes that the operations provided by all domain specifiers obey certain simple conventions designed to safeguard the information in the Library. For example, the Library assumes that all copy operations are merely observers and do not modify the value of the objects they touch. Operations not observing these conventions compromise the security of the Library and the information it contains. When a user-defined type is used as a domain type, the Library must trust the composition provided by the user to behave in a safe manner. The Library cannot guarantee that it does so.

A second disadvantage concerns Library optimization. Many database systems comparable to the Library perform optimizations based on a knowledge of what actions can result from procedure invocations. When the database system is self-contained, optimizations can be quite extensive because the database has full knowledge of the range of possible actions. The use of arbitrary types, however, introduces user-supplied procedures into the the Library. When user-supplied procedures are invoked by the Library to manipulate attribute values, the range of actions that can result is unknown, making optimization significantly more difficult. Optimization must be restricted to exclude operations by user-supplied procedures.

4.2.2.2 An Alternative: Restricting the Set of Domain Types

An alternative to allowing any type providing the operations required by the Library to serve as a domain type is to restrict domain types to a small set of predetermined types. For example, in the CLU programming environment, the set might include all the built-in types, plus a few predefined abstract types found to be especially useful for version identification, such as status types, usernames, and so on. An advantage of this approach is that compositions of the supported domain types can be built into the Library implementation. The domain specifier supplied by the user in an attribute class definition need only identify a type to model the domain. The Library itself can supply the composition.

This approach eliminates the disadvantages resulting from the use of arbitrary types as domain types. Because a composition for each of the possible domain types can be built in, the Library can guarantee that no user code can accidentally or maliciously tamper with the objects representing attribute values. Library optimization also is easier because the range of actions that can result from invoking an operation of the domain specifier is known.

We feel this approach is unsatisfactory, however, because some domains do not map easily into the provided types. Users are forced to encode the values of such domains in an unnatural form. In addition, domain checking is difficult. For domains that map naturally into the provided types, checking the composition used to create the value object serves as an adequate test that an attribute's value falls within its domain. But for many domains, composition checking is likely to be an inaccurate measure. For example, if no dates type were provided in the set of permissible domain types, a user might specify a domain representing dates with a composition of a record type containing three integers. Type and composition checking would only verify the record structure; it would fail to catch errors due to illegal values for months or days.

4.2.2.3 Library Interface Issues

The most natural representation for many attributes is in the form of enumeration types. For example, the probe sequence of a hashing abstraction could be represented by an enumeration type named "HashProbeSequence" whose value set is composed of the values "Linear," "Quadratic," and so on. An attribute could therefore be defined with name "ProbeSequence" and type "HashProbeSequence".

Although CLU offers users the ability to define any set of values as an abstract type, enumeration types are not represented conveniently in CLU. This minor inconvenience could perhaps be alleviated by providing additional support for creating enumeration types at the Library interface, but will not be addressed further in this thesis.

Another interface issue concerns literals for user-defined types. Literals provide a convenient medium for the user and the Library to communicate values of attributes. The Library prints out the values of attributes at the request of users browsing in a du; a user types in a new value to create or modify an attribute. CLU does not support user-defined type literals.

One possible solution is to add user-defined type literals to CLU. A second is to provide a facility for converting between text strings and user-defined values at the interface of the Library. Each abstraction used as the domain type of an attribute class would be required to provide operations for parsing a text string representing a value and for providing a text string representation of an abstract object. The Library could invoke these operations to interpret user-defined type literals typed by the user and to print the values of attributes.

In this thesis. we assume that some mechanism exists to support user-defined type literals. We represent such values as

<Type>#<Value>

where <Type> specifies the data abstraction whose value is being expressed and <Value> is a text string representing the literal's value.

4.2.3 Standard Vs. Customized Attribute Classes

In this section we describe the use of standard attribute classes by the CLU programming support environment, and examine the differences in the treatment of standard and customized classes. Standard attribute classes are predefined classes representing properties common to all implementations of all abstractions. Customized classes are defined by users to represent project or abstraction specific properties of implementations.

Tools in the CLU programming support environment and the conventions of the CLU development methodology require certain types of information to be available about all implementations. For example, the CLU linker must have access to information enumerating the subsidiary abstractions of an implementation being included in a composition. A subset of the standard attribute classes are used to maintain this required information in the Library.

The CLU programming support environment partitions the set of standard attribute classes into mandatory classes, recording the information needed by tools in the CLU programming support environment or by CLU methodology conventions, and optional attribute classes, expressing commonly-useful information. The Library requires every implementation to be named by an attribute from each of the mandatory classes. The optional classes are provided for the convenience of the user.

Because CLU environment tools and conventions depend on the information expressed by the standard attribute classes, the Library maintains a measure of control over the standard attributes. Attributes from the mandatory classes are protected: users are permitted to modify standard attributes only in controlled ways. Tools in the CLU environment can automatically derive the values of many standard attributes. In addition, the Library enforces consistency constraints on some standard attribute classes. For example, the user is prohibited from designating more than one implementation as the default implementation for a du.

In contrast, the Library exercises no control over customized classes. Customized attribute classes are subject only to normal authorization procedures: they are not protected by the Library. Note that this thesis does not address authorization procedures for attribute use. Neither do we propose any mechanism for allowing the user to specify consistency constraints to be applied to attributes of customized classes.

4.2.4 The Operations: Abstract Syntax and Semantics

Seven Library operations allow the user to define new attribute classes, create and modify attributes, inspect attributes, and specify contexts for resolving attribute names. This section presents the abstract syntax and semantics of these operations.

We use qualified attribute names in the definitions of all operations. Users, however, may use unqualified attribute names for customized attributes by specifying a default context. The Library automatically uses the default context to qualify all references to customized attributes. Similarly, the Library qualifies all references to standard attributes with the name of the built-in context containing the definitions of the standard classes; users may use unqualified attribute names for standard attributes at any time.

An authorization scheme is needed to control the use of these operations. This thesis, however, does not address authorization issues.

4.2.4.1 Attribute Class Definition

The Library operation DefineAttribute makes a class of names available for use by specifying their definition. Its abstract syntax is

DefineAttribute <*Qualified attribute name*> <*Domain specifier*> <*Description*>.

The qualified attribute name identifies the class of attributes being defined and has the form <Context name>:<Unqualified attribute name>. It specifies the name of the context in which the class is to be defined as well as an unqualified attribute name. <Domain specifier> is an expression denoting a composition of a data abstraction. The abstraction is the domain type of the attribute class; its composition is used to manipulate the class's value objects. <Description> is a textual explanation of the meaning of the class.

When the user invokes this operation, the Library stores a definition for the new class of attributes in the context named by the qualified attribute name. The stored definition is the tuple

[<Unqualified attribute name>, <Domain specifier>, <Description>].

It is an *error* if the context named by this operation already contains a definition for the attribute class. The operation fails if the context does not exist, if the unqualified attribute name is already defined in the named context, or if the domain specifier does not represent a composition of a data abstraction.

The following pseudo-code describes the operation.

```
If there is no context named <Context name> or if <Unqualified attribute name>
    is already defined in the context <Context name> then fail
Else store the tuple [<Unqualified attribute name>, <Domain specifier>, <Description>]
    in the context <Context name>
```

4.2.4.2 Attribute Creation and Modification

SetAttribute creates a new name for an implementation by specifying a new attribute. It may also be used to modify the value of an existing attribute. Its abstract syntax is

SetAttribute <Implementation> <Qualified attribute name> <Value>.

<Implementation> is an expression denoting an implementation. <Value> is an expression of the domain type bound to the qualified attribute name.

When the user invokes this operation, the Library evaluates the value expression to get the new attribute's value. Using the attribute definition found in the context named by the qualified attribute name, the Library performs a domain check to ensure that the expression denotes an object of the domain type created using the domain specifier. If the implementation already has an attribute with the same qualified attribute name, the existing attribute is modified to record the new value. Otherwise, the Library creates a new attribute, mapping the qualified attribute name to the value of the expression. SetAttribute fails if the context does not exist, if the attribute name is not defined in the specified context, or if the expression denoting the attribute's value fails the domain test.

The following pseudo-code describes the operation.

```
If there is no context named <Context name> or if <Unqualified attribute name> is not
    defined in context <Context name> then fail
Look up the domain specifier D bound to the name ´Unqualified attribute name>
    in context <Context name>
If <Value> does not denote an object of the domain type created with the
    the domain specifier D then fail
Let V be the result of evaluating <Value>
Store a mapping [<Qualified attribute name>, V] with <Implementation> in the Library
```

In this thesis we use the following syntactic sugar to denote an invocation of SetAttribute:

<Implementation>.<Qualified attribute name> : = <Value>.

4.2.4.3 Attribute Inspection

Three operations allow a user to inspect the attributes associated with an implementation or set of implementations.

ReadAttribute retrieves a copy of the value of an attribute on a single implementation. Its abstract syntax is

ReadAttribute <Implementation> <Qualified attribute name>

where <Implementation> is an expression denoting an implementation. ReadAttribute searches the attributes associated with the specified implementation for an attribute whose name matches the qualified attribute name. If it finds a matching attribute, the Library returns a copy of its value. ReadAttribute fails if no such attribute exists.

The following pseudo-code describes the operation.

```
If <Implementation> has an attribute with name <Qualified attribute name> then
    Return a copy of its value
Else fail
```

In this thesis we use the following syntactic sugar to denote an invocation of ReadAttribute:

<Implementation>.<Qualified attribute name>.

ReadAttributeOnSet retrieves copies of the values of an attribute on a set of implementations. Its abstract syntax is

ReadAttributeOnSet <Set of implementations> <Qualified attribute name>

where <Set of implementations> is an expression denoting a set of implementations. ReadAttributeOnSet returns a set of mappings from implementations in the set to copies of their values for the specified qualified attribute name. Implementations for which the attribute is not specified are not included in the returned mapping.

The following pseudo-code describes the operation.

```
Let M be an empty set of mappings
For each implementation I in <Set of implementations> do
    If I has an attribute with name <Qualified attribute name> then
        Add a mapping [I, copy of the attribute's value] to M
End for
Return M
```

In this thesis we use the following syntactic sugar to denote an invocation of ReadAttributeOnSet:

<Set of implementations>.<Qualified attribute name>.

ReadAllAttributes retrieves copies of all the attributes associated with a single implementation. Its abstract syntax is

ReadAllAttributes <Implementation>

where <Implementation> is an expression denoting an implementation. ReadAllAttributes returns a set of mappings from attribute names to attribute values, where each mapping corresponds to an attribute associated with the implementation.

The following pseudo-code describes the operation.

```
Let M be an empty set of mappings
For each attribute associated with implementation <Implementation> do
    Add a mapping [attribute's name, copy of the attribute's value] to M
End for
Return M
```

4.2.4.4 Context Operations

The Library operation NewContext creates a new context in which attribute classes may be defined. Its abstract syntax is

NewContext <Context name>

where <Context name> is the name to be given a new context.

The Library operation DefaultContext specifies a default context with which the Library can resolve all unqualified references to customized attribute classes. Its abstract syntax is

DefaultContext <Context name>

where <Context name> is the name of an existing context. This operation fails if no context of this name exists.

4.2.5 Summary of the Proposal

We propose an expressive naming scheme using attributes as names for implementations in the CLU Library. An attribute is a pair

[<Qualified attribute name>, <Value>]

The value may by any CLU object. A qualified attribute name consists of an unqualified attribute name and the name of a context. We use the notation

<Context name> : <Unqualified attribute name>

to denote a qualified attribute name. An unqualified attribute name is any string. This thesis assumes the existence of a user-friendly naming environment for contexts.

A qualified attribute name identifies a class of attributes; it represents a property of implementations. A class's member attributes record the behavior of individual implementations with respect to that property.

An attribute class definition binds the qualified attribute name shared by all the member attributes to the domain from which the member attributes' values are drawn. The definition is stored in the context identified by the context name part of the class's qualified attribute name. The definition consists of a mapping from the unqualified attribute name to a domain specifier and a description of the intended use of the class. A domain specifier is a composition of a data abstraction used to generate the set of objects comprising the class's domain. The abstraction implemented by the domain specifier is called the domain type.

Subject to the restriction that a domain type must provide certain operations needed by the Library to manipulate attribute value objects, any data abstraction (including a user-defined abstraction) may be selected as a domain type.

For example, an attribute [C:A,V] with qualified attribute name C:A and value object V created with composition D of type T, belongs to a class of attributes defined in context C. The definition of this class is represented by a mapping in C from the class's unqualified attribute name A to composition D of T and a description of the meaning of the class. The set of legal values for the class's attributes consists of those object that can be created with composition D.

The Library provides a set of predefined attribute classes whose definitions are found in a built-in context. These are divided into mandatory classes and optional classes. The former record information needed by the CLU programming environment and are required for all implementations. Users are also permitted to define new, customized classes of attributes.

Several operations are provided by the Library to allow users to create and inspect attributes. The abstract syntax of these operations follows. We use the notation

→ <Result>

to indicate that the operation returns the object <Result>.

```
DefineAttribute <Qualfied attribute name> <Domain specifier>
SetAttribute <implementation> <Qualified attribute name> <Value>
ReadAttribute <Implementation> <Qualified attribute name>
                          → <Attribute value object>
ReadAttributeOnSet <Set of implementations> <Qualified attribute name>
                          → <Mapping[<Implementation>,<Attribute value object>]
ReadAllAttributes <Implementation>
                          → <Mapping[<Qualified attribute name>,<Attribute value object>]
NewContext <Context name>
DefaultContext <Context name>
```

Although the operations above require the use of qualified attribute names, the Library permits a user to use unqualified names for any reference to a standard attribute class. Unqualified references are also permitted for customized attribute classes if a default context for resolving such references has been specified.

4.3 Examples

We return to the hashing abstraction of Section 3.1 to present some examples. We assume the existence of a default context named DefaultCxt containing the definitions of all customized attribute classes used in these examples.

Every implementation is automatically supplied with attributes from the mandatory predefined attribute classes. These include the attribute CreationTimestamp with domain type Timestamp, recording the implementation's creation date, and the attribute CreatedBy with domain type User, recording its creator. Additionally, the creator is asked to supply an explanation of why this implementation was created as the value of attribute Purpose. Compiling an implementation automatically applies the SubsidiaryAbstractions attribute, whose value lists the dus referred to in compilation. Similarly, BinaryUsedIn's value is the set of compositions containing the binary component of an implementation. Two implementations sharing a source component list each other as the value of the mandatory attribute SourceSharedWith. Finally, one implementation is marked as the default implementation to be used by the general public using the attribute DefaultForDU with value True. An alternatives relationship based on algorithm can be expressed with the mandatory attribute Alternative. Users supply each implementation with either the value "Buckets" or the value "Open addressing" for this attribute.

To discriminate among versions in the same alternative (sharing the same algorithm), users define customized attribute classes. The invocation

DefineAttribute HashTableSize <Domain specifier>

defines a new attribute class recording hashtable size, storing the class definition in the default context DefaultCxt. The <Domain specifier>, in this case, denotes a composition for the inteoer abstraction. A second customized attribute class, HashProbeSequence, could be defined with a user-defined enumeration type as its domain type.

Users create names for individual implementations by creating new attributes with the SetAttribute operation. The syntactic sugar

I.HashTableSize := n

represents an invocation of the operation

SetAttribute I HashTableSize n

which creates an attribute with qualified attribute name DefaultCxt:HashTableSize and value n for implementation I. Because the user specified an unqualified attribute name, the context *name is resolved to the default context.*

A user retrieves the values of attributes using the operations ReadAttribute and ReadAttributeOnSet. The invocation

ReadAttribute I HashTableSize

returns a copy of the value of the attribute named DefaultCxt:HashTableSize associated with implementation I. It can also be written as the syntactic sugar

I.HashTableSize.

If J denotes a set of implementations then

J.HashTableSize

represents an invocation of

ReadAttributeOnSet J HashTableSize

and returns a set of mappings from each implementation in J named by an attribute with attribute name DefaultCxt:HashTableSize to a copy of the attribute's value.

Figure 4-1 depicts the set of implementations for the hashing abstraction with some of the associated attributes. Note that only a subset of the implementations' attributes are shown.

4.4 The Library as a Long-Lived System

In any long-lived system, such as the Library, the question of evolution arises. Over time users create better implementations of abstractions and develop new abstractions to improve on old ones. They want to incorporate these new implementations and abstractions into their running system. Furthermore, they want to make such changes dynamically, without halting the system or losing any of its previous state. In CLU, however, dynamically replacing an implementation of a data abstraction with another implementation of the same abstraction, or replacing one abstraction with another may (or may not) yield erroneous results.

In a long-lived CLU system, state takes the form of a set of objects. If the implementation used to create one of the objects comprising a system's state is replaced with another implementation of the same data abstraction, operations of the new implementation may be used to manipulate the object created by the old implementation. This is an instance of mixing two compositions of a data abstraction, which was discussed in Sections 2.2.3 and 4.2.2. If the two implementations have incompatible expectations about the representation and invariant properties of the object, the computation will be erroneous. Sometimes, however, the expectations of the two implementations may be compatible, in which case one may safely be replaced by the other.

Similarly, a user may want to replace one abstraction with another. Using one

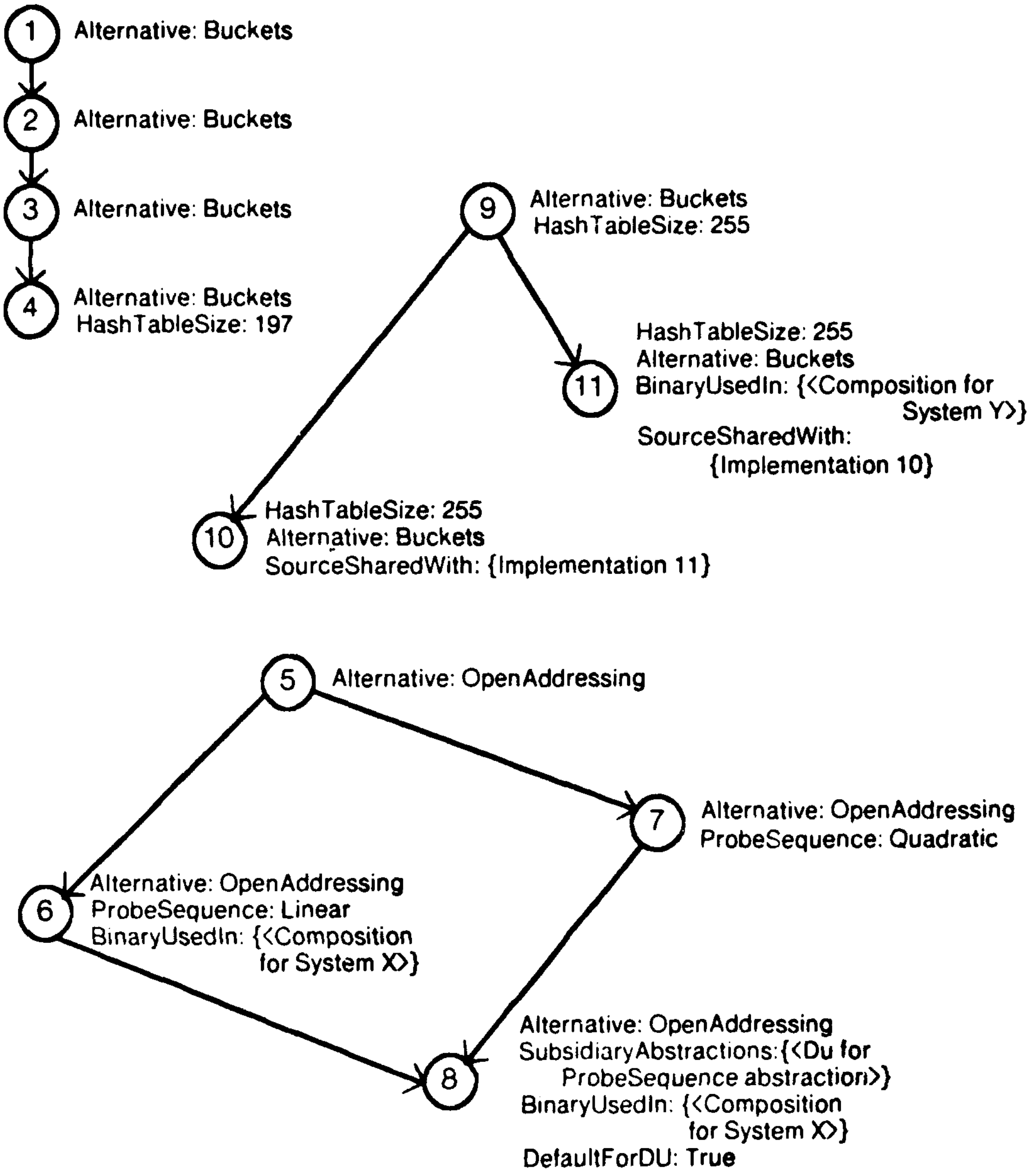

1
Alternative: Buckets
2
Alternative: Buckets
3
Alternative: Buckets
4
Alternative: Buckets
HashTableSize: 197
9
Alternative: Buckets
HashTableSize: 255
11
HashTableSize: 255
Alternative: Buckets
BinaryUsedIn: {<Composition for System Y>}
SourceSharedWith: {Implementation 10}
10
HashTableSize: 255
Alternative: Buckets
SourceSharedWith: {Implementation 11}
5
Alternative: OpenAddressing
7
Alternative: OpenAddressing
ProbeSequence: Quadratic
6
Alternative: OpenAddressing
ProbeSequence: Linear
BinaryUsedIn: {<Composition for System X>}
8
Alternative: OpenAddressing
SubsidiaryAbstractions:{<Du for ProbeSequence abstraction>}
BinaryUsedIn: {<Composition for System X>}
DefaultForDU: True

Figure 4-1: Attributes for the Hashing Abstraction

abstraction instead of another is a type error in CLU. Sometimes, however, two abstractions may be compatible and a user may want to ignore the breach of type safety. For example, a type extension could add to the set of operators provided by the original type. If the extension preserves the set of values represented by the original type and is not detectable by any of the original type's operations, then operations provided by the extension may safely be used to manipulate objects created by the original type. Other replacements, such as modifying the number of arguments in an operation's interface, may not be benign. Unfortunately, the constraints determining when it is safe to replace one abstraction with another are not well understood.

Dynamic replacement in a long-lived system impacts our naming system as follows. We represent names as attributes which are stored in the Library as mappings from attribute names to objects. Attribute value objects thus constitute state in a long-lived system. Dynamically replacing one implementation of a domain type with another, or a domain type with a new abstraction may result in errors if the Library contains pre-existing attributes (and therefore pre-existing attribute value objects) from the attribute class.

For simplicity, we prohibit modifications to domain specifications. If experience with the Library shows that users often want to replace domain specifiers, other solutions are possible. We refer the reader to [6] for a study of this problem.

4.5 Sample Built-In Attribute Classes

The Library supports both customized and predefined attribute classes. Customized attribute classes model information specific to a project, an abstraction, and so on. They are defined by the user as described earlier.

The definitions of predefined attribute classes are already known to the Library, allowing them to be used without being explicitly defined. Predefined classes are divided into mandatory attributes and optional attributes. Mandatory attribute classes encapsulate information needed by tools in the CLU programming support environment and by CLU methodology conventions. Optional attribute classes are provided as a convenience to users, encoding commonly used information such as project and subsystem membership.

In this section we suggest standard attribute classes for use in the CLU programming environment. Experience is needed to determine precisely which classes should be provided.

4.5.1 Mandatory Attributes

The mandatory attribute classes express basic information necessary to construct and use implementations, to understand the evolution of an abstraction and to maintain the du in a consistent state. For each implementation, these classes convey information uniquely identifying it, recording its use in compositions, enumerating its subsidiary abstractions, explaining how to derive or recreate it, and reporting its role in the evolution of the abstraction. Mandatory attributes are required on all implementations: every implementation must be named by an attribute from each of the mandatory classes. Mandatory attribute classes are defined in a predefined context protected by the Library.

The information encapsulated by the mandatory attributes is used by the Library and other tools in the CLU environment. To ensure the safety of this information, the Library maintains complete control over all mandatory attributes. When a new implementation is submitted, all mandatory attributes are automatically applied to it. The Library can derive automatically many of the necessary values, such as creation date. This is a convenience for the user and also ensures greater information reliability. The user is prompted for the values of attributes the Library cannot deduce, like his reason for creating an implementation. To protect the mandatory information from errors and malicious tampering, the Library performs checks for consistency and safety before permitting modification of these attributes.

A list of suggested mandatory attribute classes and their descriptions follow. Five special attribute classes, *Alternative*, *Supersedes*, *SupersededBy*, *DefaultForDU*, and *DefaultForAlternative* deserve greater attention because they play a central role in many Library operations. They are described in detail in sections 4.5.1.1, 4.5.1.2, and 4.5.1.3.

CreationTimestamp
records the date and time at which an implementation is entered into the Library. CreationTimestamp is derived automatically by the Library when the implementation is created. It may not be modified by users.

CreatedBy
records the username of the user who enters an implementation into the Library. CreatedBy is derived automatically by the Library when the implementation is created. It may not be modified by users.

CompilationTimestamp
records the date and time at which an implementation is compiled. CompilationTimestamp is recorded automatically by the CLU compiler when the implementation is compiled. It may not be modified by users.

CompiledBy
records the username of the user who invokes the compiler for an implementation. CompiledBy is recorded automatically by the CLU compiler when the implementation is compiled. It may not be modified by users.

SourceDerivation
records how the source component of an implementation is created. In particular, it names other implementations and/or tools and tool options used to derive the implementation. Tools used to create the source module automatically record this information. If no tools are used, the Library marks it as unknown. SourceDerivation may not be modified by users.

BinDerivation
records how the binary component of an implementation is created. In particular, it names the compiler and compiler options used, as well as the set equates and subsidiary abstractions available to the compiler. BinDerivation is recorded automatically by the CLU compiler when the implementation is compiled. It may not be modified by users.

SourceSharedWith

lists the set of implementations using the same CLU source component. Many different binary components can be produced from a single source component, this attribute records the set of implementations containing the same source component but different binary components. SourceSharedWith is derived automatically by the Library when the implementation is created. It may not be modified by users.

SubsidiaryAbstractions

lists the set of subsidiary abstractions used by an implementation. SubsidiaryAbstractions is recorded automatically by the CLU compiler when the implementation is compiled It may not be modified by users.

BinaryUsedIn records the set of executable programs that contain an implementation. BinaryUsedIn is maintained automatically by the Library as compositions are created. It may not be modified by users.

ImplementationNumber

is a unique identifier for an implementation. Implementations are assigned implementation numbers in order by creation timestamp, starting from one. Although the same implementation number may be used in two different dus, an implementation number is never reused within a single du. ImplementationNumber is assigned automatically by the Library. It may not be modified by users.

Purpose records the rationale for the creation of an implementation. The user provides this explanation when he enters the implementation into the Library. It may be modified at any time.

4.5.1.1 The *Alternative* Attribute

The mandatory string-valued attribute *Alternative* encapsulates the alternatives phenomenon described in Section 2.3.1. The value of this attribute names the alternative to which each implementation belongs. Users determine what criteria to use to partition the set of implementations and then assign values that reflect the chosen criteria. For example, the hashing abstraction might be split into two alternatives based on algorithm The Alternative attribute would have the value "Buckets" for implementations using external buckets and "Open addressing" for the rest.

Some dus have no obvious partitions If there are no significant differences to emphasize: all the implementations in the du are viewed as belonging to a single universal alternative

The Library asks the user to supply the name of an implementation's alternative when it is created. The Alternative attribute may be modified at any time

4.5.1.2 The *Supersedes* and *SupersededBy* Attributes

The Library makes supersession explicit through two special mandatory attribute classes Supersedes and SupersededBy whose values are sets of implementations. These attributes record the supersession relationship described in Section 2.3.2. For each implementation, Supersedes records the implementations improved upon; SupersededBy lists those that improve upon it. The Library's users determine what implementations supersede others because only users can determine improvement: improvement is not a syntactic or historical quality that can be deduced automatically by a programming support tool. *Users must state explicitly* what supersession relations are recorded in the Library. To ease the burden on the user, in many situations the Library can make common assumptions and ask the user for confirmation.

The Library enforces three restrictions on the use of these attributes. First, the Supersedes and SupersededBy attributes must be consistent across pairs of implementations. If implementation A lists implementation B in its Supersedes attribute, then implementation B's SupersededBy attribute must also include A.

Second, an implementation may only supersede (or be superseded by) implementations in the same alternative. Because two alternatives are intended for use under different conditions, it is meaningless to say that an implementation in one alternative supersedes an implementation of the other.

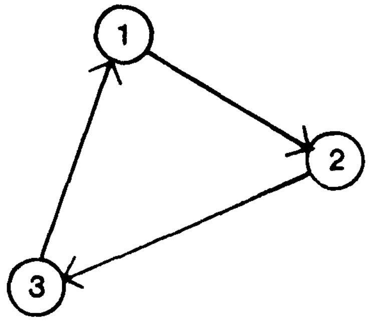


Figure 4-2:A Supersession Cycle

Finally, the Library prohibits relations in which an older implementation supersedes a more recently created one. This restriction eliminates supersession cycles. Cycles indicate an inconsistency in the supersession relations. Figure 4-2 exhibits a cycle among three implementations. Directed arcs, with the superseded implementation at the tail, mark supersession relations. By transitivity, implementation 3 supersedes implementation 1. But implementation 1 also supersedes implementation 3. Two implementations cannot each be better than the other. Although it is possible for an older implementation to improve upon a more recent one, such situations are extremely rare, resulting primarily from errors in development. Productive implementation evolution proceeds forward with time. This constraint allows the Library to prevent contradictions in supersession relations without costly checking. It is unlikely to be a major inconvenience.

In addition, the Library automatically preserves transitive supersession relationships when an implementation is deleted. Figure 4-3 demonstrates this operation. Initially, implementation 3 supersedes an implementation 1 transitively through implementation 2. If implementation 2 is deleted, all knowledge of the relationship between implementations 1 and 3 could be lost. To prevent this, the Library automatically inserts implementation 3 into the value of 1's attribute SupersededBy and implementation 1 into 3's attribute Supersedes.

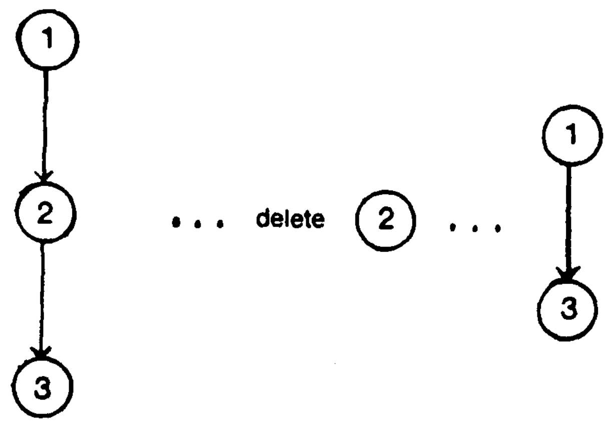


Figure 4-3: The Effect of Implementation Deletion on Supersession

4.5.1.3 The *DefaultForDU* and *DefaultForAlternative* Attributes

We identify two kinds of default implementations. One implementation may be chosen the default across all implementations of a du. In addition, one implementation may be designated the default within each alternative. Two mandatory attribute classes embody these two notions of default.

The DefaultForDU attribute identifies an implementation designated as the default across all implementations in a du. DefaultForDu is bound to boolean values; a True value marks the default implementation. At most one implementation may be chosen as the default. The Library enforces this constraint, permitting only one implementation at a time to have the value True for this attribute. All other implementations have the value False Users are permitted to modify this attribute subject to this constraint: to define a new default implementation, a user must first eliminate the old default by changing its value for this attribute to False.

The DefaultForAlternative attribute identifies a default implementation within an alternative Like the DefaultForDU attribute, this attribute takes boolean values, with a True value marking a default implementation The Library enforces two constraints on this attribute class First, each alternative may have at most one default implementation. Consequently, no two implementations within the same alternative are permitted to have the value True for this implementation. Second, the default implementation for the whole du is necessarily the default implementation for the alternative containing it; no other implementation in that alternative may have DefaultForAlternative True. The Library allows users to modify the DefaultForAlternative attribute subject to these two constraints.

4.5.2 Optional Attributes

Optional attribute classes express commonly-useful forms of information. For example, two optional attribute classes describe implementation algorithm and test status. They are provided solely for the convenience of the user, permitting them to express common forms of information without explicitly defining new attribute classes. Like mandatory attribute classes, optional attribute classes are predefined in a special built-in context.

Attributes from optional classes are maintained by the user and are not protected in any way by the library. The Library cannot derive their values and does not seek to control their usage. An authorization scheme can be used to control the manipulation of these attributes, but this thesis does not address authorization issues.

A list of suggested optional attribute classes and their descriptions follow.

Algorithm is a textual comment explaining the algorithm used by an implementation.

DataUsed is a textual comment describing the major data structures used by an implementation.

Invariant records invariant properties of an implementation.

AbstractionFunction
records the abstraction function of an implementation. It is only relevant for implementations of abstract data types.

XrefList records the use of identifiers in an implementation.

Authorization records the names of users and usergroups permitted to read, compile, or execute an implementation.

Project records the name of the project responsible for creating an implementation.

Subsystem records the name of the subsystem of a large program responsible for creating an implementation.

ReleaseNumber identifies an implementation distributed as part of an official release of some system.

OutOfDate marks an implementation no longer considered valid, for example, an implementation containing pathological bugs.

DebugStatus documents known bugs in an implementation.

TestStatus records the results of tests of an implementation.

LifeStatus records the stage of the software lifecycle represented by an implementation.

Optimization records whether an implementation is optimized and if so, whether for space or time.

SizeInBytes records the size of the binary component of an implementation.

Chapter Five

Retrieving An Implementation

The expressive naming system defined in Chapter 4 allows the user to record the properties of implementations as names. This Chapter and the succeeding one explore mechanisms using these names to retrieve implementations from the CLU Library. In this Chapter we consider the problem of selecting an implementation of a single abstraction from its du; Chapter 6 addresses the more complex problem of retrieving implementations of multiple abstractions in the context of program composition.

To select an implementation from a du, the user describes the behavior he wants the selected implementation to have. He specifies the desired properties in terms of conditions on attributes. For example, to select an implementation created by Joe after a certain date, the user specifies the attribute CreatedBy have value Joe and the value of the CreationDate attribute be greater than the specified date. To retrieve an implementation with the desired behavior, the Library tries to match the user's description against the attributes of implementations in the du.

The Library Select command embodies retrieval based on the properties of implementations. It has the form

Select <implementation-characterization>

where the <implementation-characterization>, or characterization, is the user's description of the desired behavior. Select is executed in the context of a du. It retrieves an implementation described by the characterization from the implied du. We say that the implementation retrieved *satisfies* the characterization provided by the user.

We view selection as a filtering process on the set of implementations in a du. A *filter* is a function on a set whose result is a subset of its argument set. We call the set being filtered the *input set of the filter* and the subset produced the *result set of the filter*. Select's input set is the set of implementations in a du. Its goal is a singleton set containing an implementation satisfying the characterization. It achieves its effect by applying a series of nested filters, successively reducing the number of implementations under consideration.

This chapter explores the semantics of Select. Section 5.1 proposes a simple model of selection based on an analogy to relational databases. Analysis of the model reveals three problems, discussed in Section 5.2. Sections 5.3, 5.4, and 5.5 propose extensions to the relational database approach as solutions to the three problems. Section 5.6 summarizes the semantics of a proposal combining the three extensions. The issue of missing attributes is addressed by Section 5.7. Finally, Section 5.8 presents an example using the full retrieval mechanism.

5.1 A Simple Relational Database Approach

In this section, we note an analogy between the retrieval of implementations using attributes and relational database queries. We first show how the set of implementations in a du can be modeled as a relational database. Then, using this model, we propose a simple retrieval mechanism based on relational database queries.

A relational database represents information in the form of tables [12, 15]. Each row represents an object being described in the database; columns represent properties of the objects. The set of implementations in a du can be modeled as a relational database consisting of a single table, whose rows correspond to implementations. Columns represent properties of implementations or attribute classes. Each entry in the table describes the implementation's behavior with respect to a property. In this model, a table entry represents an attribute: the entry contains the value of an attribute from the attribute class represented by the column for the implementation represented by the row.

Because every row in a table has the same number of columns, the relational database model of a du assumes that all implementations have attributes from the same classes. If any implementation has an attribute from a specific class, then all the implementations in the du must also have attributes from that class. This assumption conflicts with one of the properties of naming systems supporting implementation identification enumerated in Chapter 3. Name independence allows a user to provide a name for an individual implementation, without reference to other implementations in the du or their names. Consequently, two implementations in a du may have attributes from different attribute classes; one may not have an attribute from a class for which the other has an attribute.

A similar problem occurs in relational database systems when the value of a particular table entry is not specified. The database solution postulates an unknown value corresponding to an unspecified entry. The same mechanism can be used to preserve name independence in the relational database model of the set of implementations in a du: we postulate an unknown value corresponding to an unspecified attribute. All implementations in a du are viewed as having attributes from every class, but an attribute not explicitly provided for a given implementation has an unknown value. In the Library we represent the unknown value as a special object called *Nil*. Nil is not contained in any data type.

Using this relational database model of the set of implementations, we can pattern the Library implementation selection mechanism after relational database queries. As a simplification, we assume that all attributes are defined for all implementations; none have the special value Nil. Treatment of Nil is reserved for Section 5.7.

We turn to System R [3, 4, 5, 9] for examples of relational database queries. Because the model of a du has a single table, we consider only queries on a database consisting of one table. A System R query on a single table finds its result by two subsetting operations on the table. The first extracts a subset of the rows of the table; the second projects a subset of the columns onto the extracted rows. The query has the following format:

SELECT <*result*> FROM <*table*> WHERE <*expression*>

<Table> names the single table involved in the retrieval. <Expression> is a boolean expression, characterizing the rows to be extracted, which is evaluated once for each row in the table. When evaluating the expression for a given row, references to columns are replaced by the value of the table entry for the specified row and column. Rows for which the expression is True are extracted. <Result> specifies the information desired. It names the columns of interest, which are then projected onto the extracted rows.

For retrieval in a du, the table name and desired information are superfluous: there is only one table in the du and the result is always an implementation. Implementation selection therefore focuses on characterizing the implementations of interest. The Library retrieval operation reduces to a simplified form of the relational database operation, with a single subsetting operation selecting the rows (implementations) satisfying the characterization.

5.1.1 The Basic Proposal

A simple proposal for Select treats implementation retrieval exactly like relational database queries. In essence, the Library Select command is a filtering operation based on the characterization. The input set is the set of implementations in the du. The result set is the subset of implementations that satisfy the characterization.

Following the relational database analogy, we define the characterization to be a boolean expression. References to attribute values are encoded in the characterization as invocations to the Library attribute reading operations ReadAttribute and ReadAttributeOnSet, described in section 4.2.4. ReadAttribute returns a copy of the value of an attribute on a single implementation; ReadAttributeOnSet returns mappings from implementations to copies of their attribute values.

Select iterates over the set of implementations in the du, examining each one in turn. The implementation being evaluated is called the *current implementation*. A special symbol * is used in a characterization to refer to the current implementation. Thus, *.<attribute-name>, a sugar for **ReadAttribute(*, <attribute-name>)**, instantiates an attribute reference with the value of the attribute at the current implementation. An implementation satisfies the characterization and is retrieved if the characterization's value is True when evaluated with respect to the implementation.

The semantics of this approach to selection can be summarized by the following pseudo-code:

```
Let C be the characterization
Let the input set IS be the set of implementations in the du
Let the result set RS be the empty set
For each implementation I in IS do
    Evaluate C with * bound to I
    If C is True then insert I into RS
End For
Retrieve the implementations in RS
```

Note that the characterization can be arbitrarily complex. Because multiple implementations may share an identical attribute, a characterization specifying a single property may not identify a unique implementation. More than one property may be needed to isolate a particular implementation. A characterization specifying several properties may be viewed as a form of composite name, containing boolean subconditions representing each of the properties. For example, a user retrieving an implementation with a particular algorithm and optimized for space could specify a characterization with two subconditions joined by a boolean And operator.

5.1.2 Examples

In this section, we discuss the retrieval of implementations from a du with five implementations and three attribute classes. Figures 5-1 and 5-2 show two views of the set of implementations and their attributes. The former diagrams the du, using numbered circles to represent implementations and directed arcs to mark supersession relations. The latter depicts the single table relational database model of this set. Its rows, columns, and entries represent implementations, attribute classes, and attribute values, respectively. We use the du shown in these figures as the context for two sample invocations of Select described below.

To retrieve the implementation created by Jim after 3/13/83, the user submits the following command to the Library:

```
SELECT *.CreatedBy = User#Jim & *.CreationDate>Date#3/13/83
```

The du is implied by the context in which Select is invoked. CreatedBy and CreationDate are mandatory attributes. *.CreatedBy and *.CreationDate represent invocations of ReadAttribute, reading attribute values for the current implementation. The notation *<typename>#<text>* denotes a user-defined type literal. The operator & invokes a boolean And operator.

In response to the above command, the Library examines each of the five implementations (in arbitrary order). Evaluating the characterization for Implementation 1 invokes ReadAttribute to copy its values for CreatedBy and CreationDate. John is not equal to Jim and the date is not later than the 13th so the expression's value with respect to Implementation 1 is False. Implementation 1 does not satisfy the characterization and is not

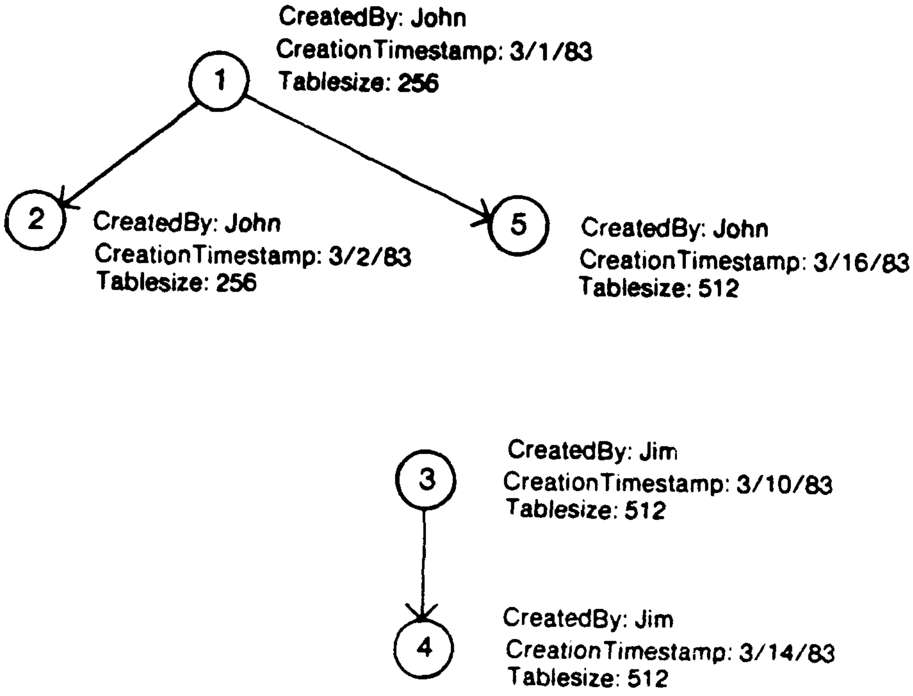


Figure 5-1:A Sample Du: Graph View

	CreatedBy	CreationDate	Tablesize
Implementation 1	John	3-1-83	256
Implementation 2	John	3-2-83	256
Implementation 3	Jim	3-10-83	512
Implementation 4	Jim	3-14-83	512
Implementation 5	John	3-16-83	512

Figure 5-2:A Sample DU: Relational Database Model

retrieved. Neither are Implementations 2, 3, or 5. Implementation 4, however, is retrieved because the characterization expression's value is True when evaluated with Implementation 4's attribute values.

In another example, the user selects implementations with the largest tablesize:

SELECT *.Tablesize = Maxsize(ℑ.Tablesize)

where Tablesize is an attribute bound to integers. ℑ denotes the set of implementations in the du. ℑ.Tablesize is a sugar for the invocation ReadAttributeOnSet (ℑ. Tablesize), returning a set of mappings from implementations in the du to copies of their values for the Tablesize attribute. Maxsize is a user-provided function, taking the set of mappings and returning the largest of the attribute values. Implementations 3, 4, and 5 are retrieved because they all have a Tablesize of 512, the largest value for Tablesize in the du.

5.2 Problems With the Relational Database Approach

In this section, we identify three problems with the simple relational database approach described above. Three succeeding sections (Sections 5.3 to 5.5) propose extensions to each of the problems we discuss here.

5.2.1 Describing a Unique Implementation

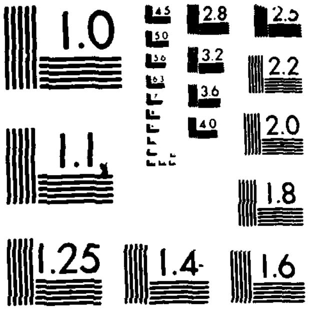
In selecting an implementation, the user specifies the properties he would like the retrieved implementation to possess. The relational database approach retrieves all the implementations providing those properties. The user's goal is to provide a characterization that uniquely describes a single implementation. However, if he provides too general or too specific a description, the relational database approach may find either multiple or zero matching implementations.

Multiple implementations are retrieved when the user specifies insufficient [illegible] to isolate a single implementation. Multiple implementations can share the same [illegible] properties. If the user specifies only properties shared by several [illegible]

AD-A142 018 NAMING IN A PROGRAMMING SUPPORT ENVIRONMENT(U)
MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER
SCIENCE J N LANCASTER FEB 84 MIT/LCS/TR-312
UNCLASSIFIED N00014-75-C-0661 F/G 9/2 NL

2/2

END
DATE
FILMED
7-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

characterization is ambiguous; several implementations may satisfy it. In this situation, a retrieval mechanism cannot recover on its own. Its only recourse is to consult with the user for further assistance in selecting an implementation.

Retrieving zero implementations, on the other hand, is a consequence of over-specification. The user has specified too many properties: no implementation possesses all of them. No implementation satisfies the user's request.

Over-specification can occur in two ways. One way marks an error condition. The user requires the retrieved implementation to provide all the specified properties. No such implementation exists. Retrieval fails.

Over-specification also occurs when the user specifies more properties than he actually requires. He specifies some inessential properties, properties that are desirable but not necessary. He may be satisfied by an implementation providing only a subset of the specified properties. Given that no implementation exists satisfying his full request, he is willing to give up some of the properties specified by the characterization. This scenario is probably very common. A user frequently views the properties comprising a characterization as having various levels of importance; he assigns differing weights to them. We expect that users will often specify some properties describing absolute requirements, others that he views as important but not critical, and still others that are not particularly important by themselves, but are primarily useful to disambiguate among the subset of a du's implementations satisfying other more important properties.

For example, although he provides a characterization with properties specifying target machine and optimization requirements, a user may find unoptimized implementations targeted to his machine acceptable. He requires that the target machine condition be met, and would prefer that both properties were satisfied, but might be willing to forego optimization if no implementation exists with satisfying both conditions. Optimization is a secondary concern, of interest only if there exist several implementations targeted to the specified machine.

To support the user in isolating a single implementation, a retrieval mechanism should allow the user to express his perceptions of the relative weights of properties, and to identify less important properties that can be eliminated if no implementation satisfies the full characterization. In this way, the user can supply detailed descriptions, without risking failure due to over-specification. A less important property can be used to select among the implementations satisfying more important properties, but can be ignored or eliminated if no implementations satisfy both the less important property and more important properties. If no implementation satisfies the characterization, the retrieval mechanism can try to select an implementation using a subset of the characterization's properties, eliminating properties identified by the user an inessential. The user's view of the relative importance of properties determines what subset of the properties to use.

The relational database approach cannot provide the above support because it uses a single monolithic characterization. The individual properties specified by the user are not represented explicitly as subconditions, but instead are buried within the characterization expression as boolean subexpressions. They cannot be identified independently, assigned relative weights, or selectively eliminated from the characterization. All properties are viewed as having equal weight; none are considered inessential.

5.2.2 Supporting Standard Practices

A second failing of the relational database approach is that it does not incorporate mechanisms commonly used in implementation identification. Existing programming environments use several standard mechanisms for retrieving implementations. For example. one mechanism embodies the notion of defaults. One implementation is designated as the standard, to be retrieved whenever a user requests an implementation, but does not identify a particular implementation. Another common practice is based on the observation that users seldom select an implementation that has been superseded by another. Usually, if a user has a choice between two implementations, only one of which has been superseded by another implementation, he chooses the unsuperseded version. Most current programming environments support this practice using creation order to approximate supersession relations, retrieving the most recently developed implementation satisfying the user's requirements.

Because these practices are so widely supported by other programming environments and have been found to be extremely useful, we believe they should be incorporated into the retrieval mechanism for the CLU programming environment. They are used so frequently that the user should not have to specify them explicitly. For example, the retrieval mechanism should adopt the common convention that unless the user explicitly disagrees, an unsuperseded implementation is more desirable than a superseded but otherwise equally acceptable implementation.

The relational database approach to retrieval does not incorporate these standard practices. It supports them only to the extent that users can explicitly encode standard practices as part of the characterization. For example, the user could use the convention selecting unsuperseded implementations over superseded ones by specifying a condition on the mandatory SupersededBy attribute as part of the characterization. This approach is inadequate because it requires extra work on the user's part to express practices used in almost every retrieval operation.

5.2.3 Efficiency

The relational database approach of evaluating the characterization once for each implementation in the du can be very inefficient under certain conditions. For example, if the user specifies the characterization

*.Size = Minsize(ℐ.Size)

where ℐ.Size invokes the operation ReadAttributeOnSet to collect the values of the Size attribute on all implementations in the du, and Minsize is a user-provided function that takes the set of implementation/attribute value mappings returned by ReadAttributeOnSet and returns the smallest of the attribute values, then the smallest size is calculated over and over, even though its value does not change.

The inefficiency is due to the fact that the characterization contains a subexpression not dependent on the current implementation. Because each iteration introduces a new current implementation, values dependent on the current implementation, like attribute references, vary as the selection mechanism iterates over the set of implementations in a du. Computations not dependent on the current implementation, however, are constant over all iterations. It is inefficient to recompute them unnecessarily.

5.3 Support for Composite Characterizations

In this section we present an extension to the relational database approach that addresses the first of the three problems enumerated by Section 5.2. The problem concerns finding the correct level of detail for describing a unique implementation. A general description may be ambiguous, matching multiple implementations; a detailed description specifying more properties, however, may result in over-specification, causing no satisfactory implementations to be found.

Our solution replaces the monolithic description of the relational database approach with a composite characterization composed of subconditions called *criteria*. The composite form allows users to express their perceptions of the relative weights of properties comprising the characterization, and to designate less important properties that can be ignored if no implementation satisfies the complete characterization. Support for detailed descriptions containing properties identified as inessential helps the user in describing a unique implementation because the inessential properties are considered only to help disambiguate among otherwise acceptable implementations. They cannot cause failure due to over-specification.

Each criterion contain.' a subset of the properties specified by the characterization. There are two kinds of criteria. One single criterion may be designated the *requirements criterion* of a characterization. A requirements criterion specifies minimum conditions that must be satisfied by any acceptable implementation. All other criteria are called *preference criteria*. Preference criteria express properties that are desirable but not necessary. They represent inessential properties that can be ignored if no implementation can be found satisfying all the properties specified by the characterization. Preference criteria are used to discriminate among the implementations satisfying the requirements criterion.

For example, a user requiring an implementation targeted to a particular operating system could request particular algorithms and/or performance properties as preference criteria. The requirements criterion would specify the target operating system. If only one implementation targeted to the appropriate operating system exists, it is retrieved regardless of whether or not it satisfies the preference criteria. If multiple implementations satisfy the requirements criterion, but none satisfy all the preference criteria as well, then a subset of the preferences is used to select an implementation from those that satisfy the requirements criterion.

A priority mechanism on criteria allows users to express the relative importance of properties. Each criterion corresponds to a level of priority. Properties viewed by the user as having equal weight can be combined in a single criterion; properties of differing weights are assigned to distinct criteria. Note that only one requirements criterion is needed because all required properties are equally important. Inessential properties may not all be of equal weight, however, so multiple preference criteria may be needed. Preference criteria are ordered by the user to reflect the relative weights of the properties they express. By definition, the requirements criterion is of greater priority than any of the preference criteria.

By assigning priorities to criteria, the user controls which subset of the properties expressed by a characterization should be used to select an implementation if no implementation satisfies the complete characterization. Properties expressed by higher priority criteria are considered before those expressed by lower priority criteria.

5.3.1 A Proposal For Composite Characterizations

We extend the relational database approach to support detailed descriptions containing identifiably inessential properties as follows. A characterization is defined to be a composite form of description partitioned into subconditions called criteria. It contains an arbitrary number of preference criteria ordered by decreasing priority and at most one special criterion designated a requirements criterion. The requirements criterion, if it exists, is of greater priority than all of the preference criteria. Like the characterization of the relational database model, each criterion is an arbitrary boolean expression used as a filter. References to attribute values are encoded as calls to the Library attribute reading functions, ReadAttribute and ReadAttributeOnSet.

The selection mechanism uses a model of repeated subsetting, with each of the characterization's component criteria corresponding to one subsetting operation. Each criterion is used as a filter to disambiguate among the set of implementations satisfying the previous criteria, reducing the number of implementations still under consideration. Criteria are applied in order of decreasing priority, with the result set of one criterion serving as the input set of the next. Thus, the requirements criterion is applied to the set of implementations in a du, yielding the set of implementations satisfying the requirements. The first preference criterion filters the result set of the requirements criterion. Each succeeding criterion filters the result set of the previous criterion.

Figure 5-3 traces selection of implementations using a characterization comprising three criteria. The outermost circle (labeled 0) circumscribes the set of implementations in a du and serves as the input set for the highest priority criterion. The circle labeled 1 is its result set and also serves as the input set to the criterion of next highest priority. The circle numbered 2 is the result set of this criterion and the input set to the remaining criterion. Finally, as the result set of this final criterion, the innermost circle circumscribes the set of implementations retrieved.

Each subsetting operation corresponds to a single criterion. We call the criterion being processed the *current criterion*, and its input and result sets are the *current input set* and *current output set*. In contrast to the relational database model, the input set of a criterion is not predetermined. In the relational database model, the input set of the single expression characterization is the set of implementations in the du and can be referenced in the characterization as $\mathcal{I}$. In this more complex approach, however, the input set of a criterion is computed based on previous criteria. There is no way to statically name the current input set. The special symbol ** can be used in a criterion to refer to the dynamically determined current input set.

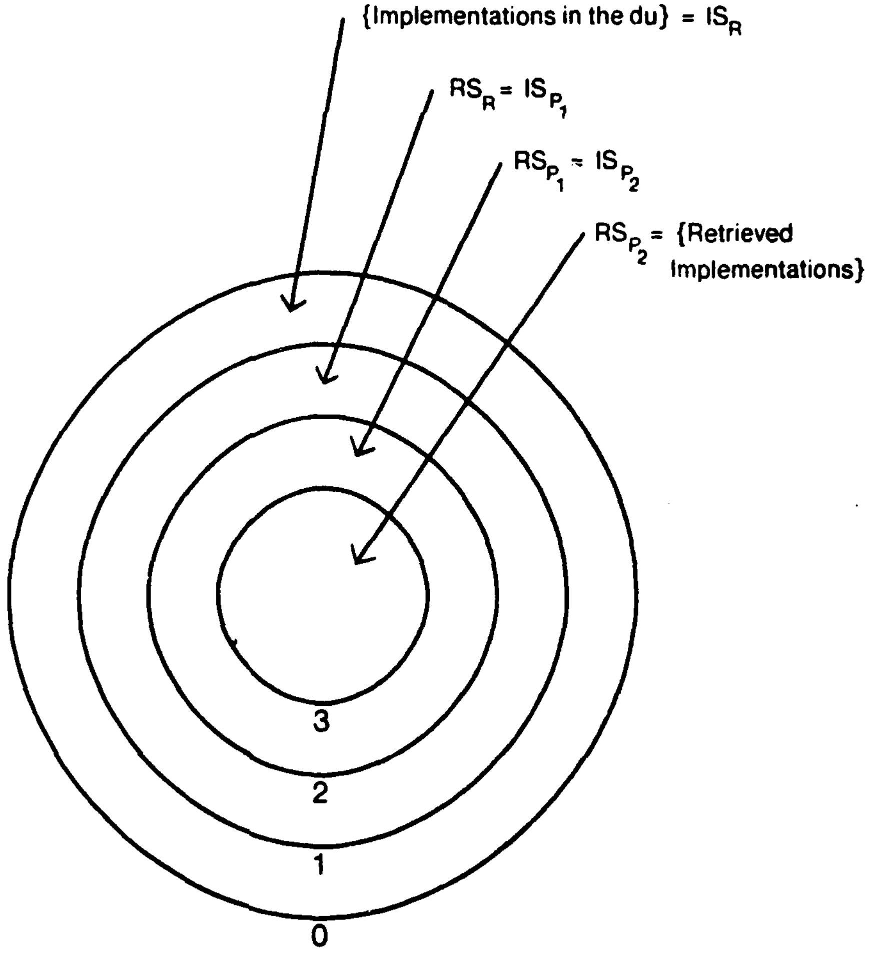

{Implementations in the du} = IS_R
$RS_R = IS_{P_1}$
$RS_{P_1} = IS_{P_2}$
RS_{P_2} = {Retrieved Implementations}
3
2
1
0

Figure 5-3:Tracing a Selection Operation

Each subsetting operation has two parts. The first is a filtering operation on the current input set using the current criterion. This operation is completely analogous to filtering the set of implementations in a du using the characterization in the relational database approach. The current criterion is a boolean expression, evaluated once for each implementation in its input set. The implementation being evaluated is called the current implementation and can be referred to in the current criterion with the special symbol *. An implementation satisfies the current criterion and is included in the current result set if the criterion's expression is True when evaluated with respect to the implementation.

The second part of the subsetting operation tests the current result set. If the current result set is not empty, then the subsetting operation yields the current result set. If it is empty, we say the current criterion is *void*. The Library's treatment of a void criterion depends on whether the criterion specifies requirements or preferences.

Selection fails if the result set of the requirements criterion is empty. The requirements criterion specifies minimum standards for accepting an implementation. If no implementation meets these standards, the Library cannot possibly select one.

Preference criteria, on the other hand, are only guides for narrowing down the number of choices. They express inessential properties. Presumably all the implementations that satisfy the requirements criterion are acceptable to the user; the preference criteria merely help to find the most attractive of the acceptable alternatives. A void preference criterion should not cause the selection operation to fail. An empty result set for a preference criterion merely indicates that the particular criterion cannot contribute to determining the best choice of implementation.

In our proposal, a void preference criterion is ignored. It has no effect on the selection process. No subsetting occurs as a result of a void preference criterion. Instead, the current input set is passed along as the current result set, as if all of its implementations had satisfied the void criterion. Thus, the input set of the succeeding criterion is the same as the input set of the void criterion.

Note that the implementation forming the singleton result set of any criterion is retrieved regardless of whether or not the criterion is the last in the characterization. For each succeeding criterion, either the implementation satisfies the criterion, in which case its result set is the singleton set, or the criterion is void. Since no criteria precede the requirements criterion, the succeeding criterion must be a preference, so the singleton input set is substituted for an empty result set. In either case, the succeeding criterion rubberstamps the implementation, passing the singleton set along as its result set.

If the result set of the lowest priority criterion is not a singleton set, the Library is unable to retrieve a unique implementation. Its only recourse is to consult with the user.

5.3.2 Summary

A characterization is an ordered set of components called criteria. The order defines the relative priority of criteria. The highest priority criterion may be designated a requirements criterion; all others are preference criteria. Each criterion is an arbitrary boolean expression.

The following pseudo-code summarizes the semantics of our proposal for implementation retrieval using composite characterizations.

```
% Requirements criterion filter
Let C be the characterization
Let IS be the set of implementations in the du
Let RS be the empty result set
For each implementation I in IS do
    Evaluate the requirements criterion R of C with * bound to I,
        ** bound to IS
    If R is True then insert I into RS
End For
If RS = the empty set then selection fails

% Preference criteria filters
For each preference criterion P in C do
    Let the input set IS := RS
    Let IRS be an empty intermediate result set
    For each implementation I in IS do
        Evaluate P with * bound to I, ** bound to IS
        If P is True then insert I into IRS
    End For
    If o(IRS) > 0 then RS := IRS
    Else RS := IS
End For

% Check for non-unique result
If o(RS) > 1 then consult with the user
Else retrieve the implementation in RS
```

The first half processes the requirements criterion, collecting the acceptable implementations in the result set RS and failing if no implementations meet the requirements. The second half iterates through the preference criteria. The variable IS, initialized with the result set of the requirements criterion, is updated to reflect each preference criterion applied.

5.3.3 Examples

Figure 5-4 depicts the set of implementations in a sorting abstraction and some of the implementations' attributes.

A selection command with the characterization

Requirements:	*.TargetMachine = Machine#Vax
Preferences:	*.Color = Color#Red
	*.Project = Project#QRS

selects implementations 2, 4, and 5. The requirements criterion eliminates implementation 3, leaving implementations 1, 2, 4, and 5 as the set of acceptable implementations to be selected among by the preferences. The first preference, *.Color = Color#Red, is applied to those four implementations. None of them, however, have Color equal to Red, so this criterion's intermediate result is void. Its input set is substituted for its empty result set. It does not contribute any information to disambiguating among the implementations satisfying

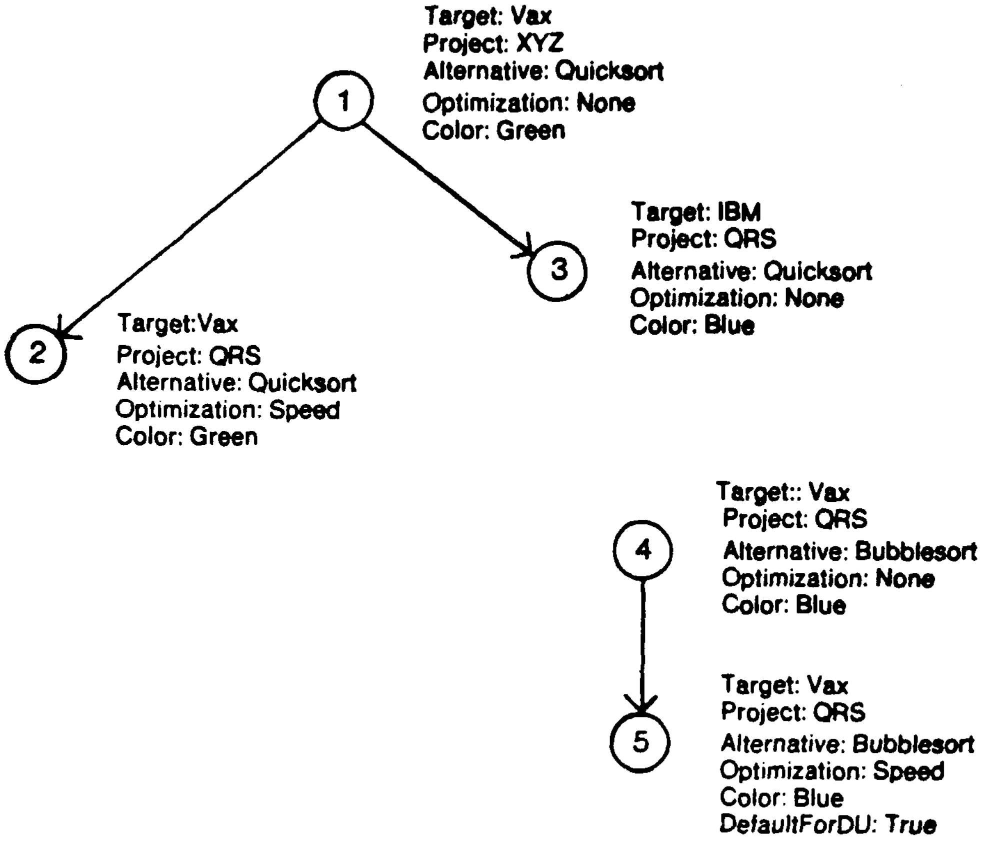


Figure 5-4:A Sorting Abstraction

the criteria of higher priority, so it is ignored. The second preference is applied to the same set of implementations. Of these, implementations 2, 4, and 5 satisfy the criterion. Because this is the last preference, the Library must consult the user to determine which of these implementations to retrieve.

Suppose the user adds more preferences, making the characterization

Requirements:	*.TargetMachine = Machine # Vax
Preferences:	*.Color = Color # Red

*.Project = Project#QRS
*.Alternative = "Quicksort"
*.Optimization = Optimization#Speed

Implementations 2, 4, and 5 form the input set to the third preference criterion. Its result set is a singleton, consisting of implementation 2. The last preference criterion is void, so its input set replaces its empty result set. Thus, it rubberstamps the single implementation it received. Implementation 2 is retrieved.

5.4 Incorporating Common Practices

The second of the three problems identified in Section 5.2 is that the relational database approach does not provide adequate support for practices commonly used in implementation retrieval. Although these practices have consistently been found very useful, the relational database approach provides no convenient mechanism for including them in implementation identification operations.

We propose that commonly used practices be incorporated directly into the retrieval mechanism. Because the common case has the user employ these practices, we propose that they be invoked automatically by the retrieval mechanism, relieving the user of the burden of explicitly invoking them himself. Library options can be provided to allow users the ability to suppress invocation in the rare cases when he does not want to observe these practices.

In this section, we identify two classes of mechanisms that are commonly used in implementation retrieval. We propose three filtering operations based on mandatory attributes that support these practices. Subject to user control by Library options, these filters are used to select among the set of implementations satisfying a characterization.

Thus, they constitute three additional subsetting operations applied to the result set of the last criterion in the characterization.

Note that the filters we propose are only examples. We have chosen filters that embody two common practices. If experience shows that other conventions are observed frequently by users retrieving implementations, then new automatic filters could be added to incorporate these practices.

5.4.1 Using Supersession Relations

The supersession filter is based on the observation that users seldom select an implementation that has been superseded by another. It reflects a common practice in which most current programming support environments, using creation order to approximate supersession, automatically supply the most recent version. Taking the characterization's result set as its input set, the supersession filter eliminates implementations superseded by others in the input set. One implementation can supersede another either directly or transitively through another implementation or implementations. If the supersession filter's input set contains an implementation superseded by another, the superseded implementation is eliminated from consideration.

Throughout this thesis, we have displayed the set of implementations in a du as a directed graph, with the arcs marking supersession relationships. Using this view of a du, a characterization's result set is the subgraph composed of those nodes satisfying the characterization. Figure 5-5 depicts the result set of some characterization. Continuous arcs represent explicit supersession relations. Dotted arcs represent supersession relations derived by transitivity through an implementation or implementations not contained in the result set of the characterization. The supersession filter eliminates all non-leaf nodes from the set of implementations under consideration. Its result set contains multiple implementations if the supersession graph includes branches or disjoint subtrees.

A Library option is provided to give the user explicit control over the supersession filter. If the option is set, the filter's result set contains the leaf nodes as described above. If not, the result set is defined to be the entire input set (i.e., the result set of the characterization). Because cycles in the supersession relation are prohibited by the definitions of the supersession attributes, it is impossible to get an empty result set from this filter, given an non-empty input set.

Returning to the Figure 5-5, we see that the characterization's result set consists of implementations 1 through 3, 7, 8, 10, and 12. Implementation 10 supersedes implementation 1 transitively through at least one implementation not contained in the figure. Similarly, implementation 12 supersedes implementation 8. If the Library option is set to allow filtering based on supersession, the result set of the supersession filter contains the three leaf nodes, implementations 3, 10, and 12. If the option is not set, non-leaf nodes are returned as well; the result set is the full graph shown in Figure 5-5.

5.4.2 Using Defaults

After the supersession filter, two filters based on defaults are applied. Both are subject to user control via Library options. Because of constraints imposed by the Library on the mandatory attributes used by these filters, these two filters yield the same result when applied in either order: they are commutative. For simplicity, we assume the filter using the DefaultForDU attribute is applied first.

The first filter uses the DefaultForDU attribute and takes the result set of the supersession filter as its input set. If the option controlling this filter is set and the input set

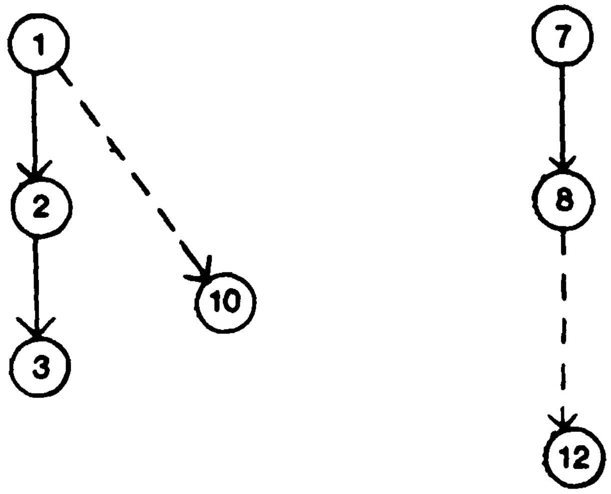


Figure 5-5:Supersession's Role in Selection

contains the implementation whose value for the mandatory attribute DefaultForDU is True, then the result set is the singleton set consisting of that implementation. Otherwise, the filter's result set is the same as its input set.

The second filter is applied to the result set of the DefaultForDU filter. If all the implementations in its input set are part of the same alternative, it selects the default implementation for the alternative. More precisely, if the option controlling the filter is set, and all the implementations in its input set share the same value for the mandatory attribute Alternative, and the input set contains an implementation whose value for the mandatory attribute DefaultForAlternative is True, then the result set is the singleton set consisting of that implementation. Otherwise, the filter's result set is the same as its input set.

If applying the three automatic filters does not yield a unique implementation, the user is consulted. At this point, the Library has no more information that can be used to deduce the user's intention. Only the user can determine which of the several implementations satisfying the characterization. supersession. and defaults filters best suit his needs.

5.5 Optimization

The third problem identified in the relational database approach is inefficiency. Computations not dependent on the current implementation are repeatedly performed, once for each implementation being evaluated. This recomputation is unnecessary because only values dependent on the current implementation should change.

Searching for a solution, we turn again to the example of System R. System R queries limit the functions that can be invoked from the boolean expression corresponding to the characterization to a small set of predefined functions. Because the behavior of all these functions is known, System R can safely perform optimizations eliminating superfluous recomputations.

Our proposal, however, allows arbitrary invocations to appear in the characterization expression. Because the behavior of arbitrary invocations cannot be predicted, optimizations cannot be done safely. In particular, it is not safe to optimize characterizations invoking functions with side effects. The result set of the characterization invoking the function having side effects only once may be different from the result set if the function is invoked repeatedly.

To support optimization and eliminate unnecessary computation, the Library selection mechanism prohibits side effects from invocations in the characterization. Without side effects, the Library can isolate subexpressions not dependent on the current implementation, compute their values once, and use the values to evaluate the characterization with respect to each implementation. Although this places a restriction on the functions that can be invoked in the characterization, we believe it is justifiable, not only for the sake of efficiency, but also because side effects are undesirable here. Selection should be a benign operation, evaluating each implementation solely on the basis of its properties. The decision to select a given implementation should not depend on whether or not other implementations have already been evaluated.

Note that because the current input set of a criterion is computed dynamically using the higher priority criteria as filters, subexpressions of a criterion not dependent on the current implementation but dependent on the current input set cannot be computed until all the higher priority criteria have been processed. Subexpressions not dependent on either can be computed before any of the characterization has been applied.

5.6 Summary of Selection Semantics

Combining the mechanisms proposed in sections 5.3, 5.4, and 5.5 leads to the selection process summarized below.

```
% Requirements criterion filter
Let C be the characterization
Let IS be the set of implementations in the du
Let RS be the empty result set
Let $E_1$, $E_2$,... be the values of any subexpressions of the
    requirements criterion R of C not dependent on the current
    implementation
For each implementation I in IS do
    Evaluate R with * bound to I, ** bound to IS, and using the
        values $E_1$, $E_2$,... as necessary
    If R is True then insert I into RS
End For
If RS = the empty set then selection fails

% Preference criteria filters
For each preference criterion P in C do
```

```
    Let the input set IS : = RS
    Let IRS be an empty intermediate result set
    Let $E_1$. $E_2$,... be the any subexpressions of P
        not dependent on the current implementation
    For each implementation I in IS do
        Evaluate P with * bound to I, ** bound to IS, and using
            the values $E_1$, $E_2$,... as necessary
        If I P is True then insert I into IRS
    End For
    If o(IRS) > 0 then RS : = IRS
    Else RS : = IS
End For

% Supersession filter
If supersession option set then
    IS : = RS
    For each implementation I in IS do
        If I is superseded by any other implementation in IS then
            Delete I from RS
    End For

% DefaultForDU filter
If default for du option set then
    IS : = RS
    For each implementation I in IS do
        If I.DefaultForDU = True then
            RS : = the singleton set containing I
    End For

% DefaultForAlternative filter
If default for alternative option on then
    IS : = RS
    For each implementation I in IS do
        If I.DefaultForAlternative = True then
            RS : = the singleton set containing I
    End For

% Check for non-unique result
If o(RS)>1 then consult the user
Else retrieve the implementation in RS
```

5.7 Coping With Unspecified Attribute Values

The relational database model that is the basis of our proposal requires each implementation to have an attribute belonging to every attribute class for which another implementation in the du has an attribute. This requirement conflicts with the goal of name independence, allowing users to provide an attribute for one implementation without considering other implementations in its du. In Section 5.1, we borrowed a standard relational database solution to this problem and introduced a unknown attribute value, represented in the Library by a special object called Nil. The unknown value acts as a placeholder for the values of unspecified attributes. By substituting for values of attributes not explicitly provided by the user, Nil supports the relational database constraint requiring every implementation to have an attribute from every class.

Up to this point in the chapter, we have ignored the question of unspecified attributes. Our proposal was based on the simplifying assumption that no attributes were Nil-valued. In this section, we consider the consequences of supporting attributes with unknown values, and propose an extension to our retrieval mechanism to handle references to Nil-valued attributes in characterizations.

Nil-valued attributes impact implementation retrieval because they indicate missing information. An attribute records an implementation's behavior with respect to some property. If an attribute is Nil-valued, then its behavior with respect to that property is unknown. How can an implementation be evaluated if a characterization specifies a condition on a property for which the implementation's behavior is unknown?

To determine what behavior is desirable, we consider three examples. The first examines a simple criterion, the others more complex criteria composed of multiple properties joined by boolean operators. The examples are based on a du with four implementations and two of their attributes, shown in Figure 5-6. CreatedBy is a mandatory attribute bound to usernames; Subsystem is a string-valued optional standard attribute. Note that implementation 2 does not have attribute Subsystem; its value is Nil. Figure 5-7 depicts the relational database model of this du.

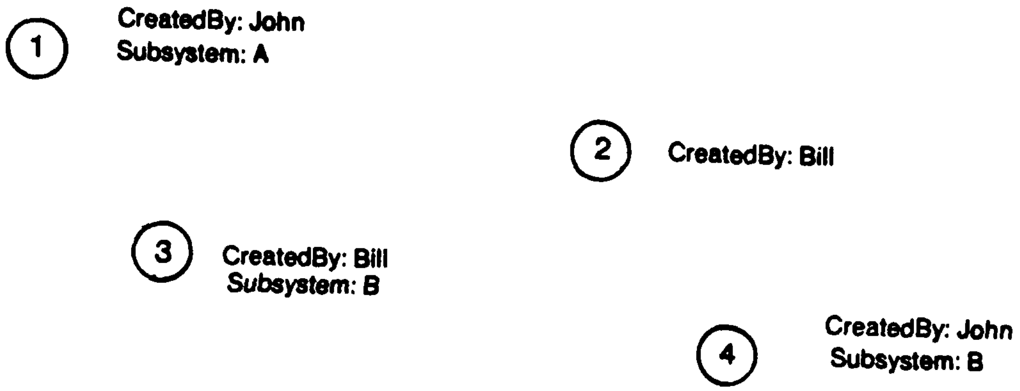


Figure 5-6: The Nil Attribute Value: Graph View

	CreatedBy	Subsystem
Implementation 1	John	A
Implementation 2	*Bill*	⊥
Implementation 3	Bill	B
Implementation 4	John	B

Figure 5-7: The Nil Attribute Value: Relation Database View

For the simple criterion

*.Subsystem = "B"

the Library should retrieve only Implementations 3 and 4. Implementation 1 fails because it has the wrong subsystem. Implementation 2, however, has no subsystem attribute. With no information available, the Library must assume that the implementation does not satisfy the criterion.

The criterion

*.Subsystem = "B" & *.CreatedBy = User#Bill

cannot be evaluated for Implementation 2 because it has a Nil value for the Subsystem attribute. One plausible interpretation is that Implementation 2 is not part of Subsystem B or else someone would have recorded the fact in an attribute. The user does not want Implementation 2 because it does not have a value for one of the conditions he is interested in.

On the other hand, for

*.Subsystem = "B" | *.CreatedBy = User#Bill

one could imagine the user is someone who can use any implementation created either by Bill or for subsystem B. An acceptable implementation need not have both attributes. The user would want to select both Implementations 2 and 3, despite the fact that Implementation 2 does not have a Subsystem attribute.

The three examples offer conflicting views of "correct" behavior in the presence of Nil. The first and second assume a Nil value for a referenced attribute automatically eliminates an implementation from consideration. A Nil value fails every condition. This is the approach taken by relational databases. The third example, however, takes a more liberal view, treating Nil as an unknown but not necessarily pathological quantity. It ignores the Nil value, using other subconditions to evaluate the implementation.

Either perspective is reasonable. The Library should support both, allowing the user to determine which is appropriate to a given situation.

5.7.1 Treatment of Nil in System R

In System R, the operations that can be invoked in the characterization expression are limited to a predefined set of operations. They include boolean operations, arithmetic operations, comparison operators and a few special operators like max and count. Treatment of Nil is included in the specification of these functions. For example, an arithmetic comparison operator takes two integers and returns a boolean. But if one or both of its inputs denotes an attribute whose value is Nil, it returns False.

The characterization expression may take on any of the values True, False, or Nil. A row is extracted only if the expression's value is True with respect to it.

5.7.2 A Goal for the Library

Unlike System R, characterization expressions in the Library selection mechanism may invoke arbitrary functions. Invocations are not restricted to a predefined set. Adopting System R's solution would mean requiring any function invoked from a characterization and depending on an attribute value to be prepared for Nil arguments. Because Nil is a special value not contained in any type, functions used in characterizations would therefore have to be written with arguments of union types (uniting Nil with T for an argument of type T).

This approach is unacceptable for a number of reasons. First, it unreasonably burdens the user, requiring him to determine for each function invoked in a characterization what behavior is desired in the presence of Nil arguments. Second, it complicates the code he must write. Each function must not only perform its normal computation but must also handle exceptional conditions due to Nil arguments. In particular, the user must provide two functions if different circumstances require a different response to Nil-induced exceptions. Finally, since many attributes will be bound to built-in types, the built-in types must be modified to accept Nil.

Our goal is to contain Nil at the Library interface. User functions should not have to cope with Nil; they should be able to assume that, if they get invoked, all their arguments are of the correct type. The Library can "trap" Nil values when the attribute reading operation returns. User functions depending on an attribute reference should only be invoked if the value is not Nil.

5.7.3 Evaluating Implementations in the Presence of Nil

In this section, we propose an extension to our retrieval mechanism that allows implementations to be evaluated despite missing information and Nil attribute values. The basis of our proposal is a modified definition for criteria, using an expression in three-phase logic, with values True, False and Nil. Three-phase logic operators joining subconditions represented by boolean expressions act as *firewalls, controlling the propagation* of Nil between subconditions. An implementation satisfies a criterion if the expression's value is True with respect to it. Neither False nor Nil are acceptable.

To prevent users from having to deal with Nil, we say all functions preserve Nil. If any of their inputs are Nil, then all of their outputs are automatically assumed to be Nil. This allows the Library to trap Nil-valued arguments and avoid invocation of functions expecting non-Nil arguments.

When the Library *encounters* an invocation to the attribute reading operation ReadAttribute, it checks to see if the attribute has been applied to that implementation. If it has, the Library makes a copy of the value and continues to evaluate the expression. If it has not, the value of the attribute is Nil. The outputs of all functions depending on the attribute value are assumed (without execution) to be Nil. For example, if the attribute A were not applied to an implementation, then its value would be Nil. In evaluating the criterion expression

*.A = 6 & *.B = 7

the Library traps the Nil value of attribute A in the reading operation *.A. Nil is an input to the integer Equal operation, so its result is assumed to be Nil. Preserving Nil, the boolean And operation also returns Nil, yielding a Nil result for the expression and eliminating the implementation from selection.

5.7.3.1 Three-Phase Logic Operators

Three special operators, Intersect, Union, and Tilda, are provided to manipulate three-phase logic values. They are the only operations that do not preserve Nil. Treatment of Nil is explicitly a part of their specification and they are executed regardless of whether or not an argument is Nil.

The three operators parallel the boolean operators And, Or, and Not, exhibiting precisely the same behaviors for True and False operands. Nil represents an unknown truth value. It is assumed to be either True or False, but it is impossible to know which. Three-phase operations involving Nil can be evaluated by converting them to boolean expressions and instantiating Nil with True and False. If the corresponding boolean operation has the same truth value result when Nil is replaced with both True and False, then the three-phase expression has the same truth value result. If instantiating Nil with True and False yields different values, then the three-phase expression has the unknown truth value, Nil. For example, False Intersect Nil is False because both False And True and False And False yield False. True Intersect Nil, however, yields Nil since True And True does not equal True And False.

Intersect, Union and Tilda, written ∩, ∪ and ~, correspond to boolean And, Or and Not, respectively. The complete truth tables for the three-phase logic operators follow.

∩	T	F	⊥
T	T	F	⊥
F	F	F	F
⊥	⊥	F	⊥

∪	T	F	⊥
T	T	T	T
F	T	F	⊥
⊥	T	⊥	⊥

	~
T	F
F	T
⊥	⊥

5.7.3.2 User Control of Nil

The three-phase operators allow users to control the effect of Nil in criteria by controlling propagation of Nil from one subcondition to another. In an expression consisting of two subconditions joined by a three-phase logic operator, a Nil value for one propagates beyond the subcondition only if the truth value of the expression cannot be determined from the other subcondition. The three-phase logic operator isolates the effect of Nil to the subcondition in which it appears. For example, a merge operation evaluates each implementation independently with respect to each of the subconditions. An implementation satisfies the expression if either of the two subconditions are True. The criterion

$$*.A = 6 \cup *.B = 7$$

retrieves an implementation with a value of 7 for attribute B but for which attribute A had not been applied at all. The Library traps Nil at the reading operation *.A, forcing a Nil result for the dependent integer equal operation. The union operation, however, acts as a firewall against that Nil, preventing it from propagating farther in the expression. Because the other subexpression, *.B = 7, is True, the implementation satisfies the criterion.

Both three-phase logic operators and boolean operators are needed to offer the user complete control over the effect of Nil. boolean operators are used to join two subconditions viewed by the user as being integrally bound together; three-phase logic operators join subconditions that may be evaluated independently and may compensate for each other. We return to the examples of Section 5.7 to demonstrate use of the two kinds of operators.

To get the strict behavior shown in the second of the three examples, the user uses boolean operators:

*.Subsystem = "B" & *.CreatedBy = User#Bill

Implementation 1 fails because it has the wrong Subsystem value; Implementation two fails because the expression evaluates to Nil as a result of the missing attribute; Implementations 3 and 4 satisfy the criterion.

To get the more liberal behavior of the third example, the user uses three-phase logic operators:

*.Subsystem = "B" ∪ *.CreatedBy = User#Bill

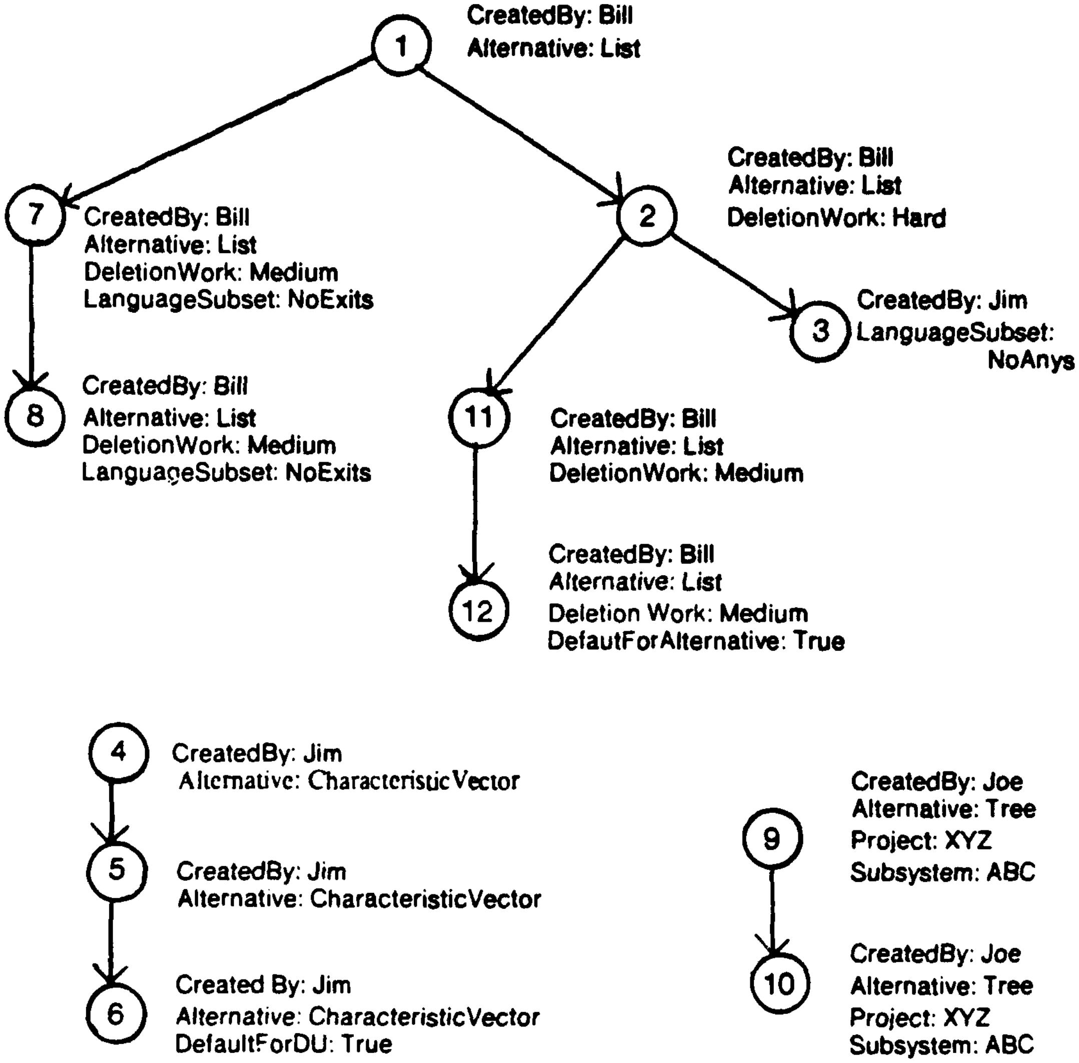
Implementation 1 fails because it has the wrong Subsystem, but Implementations 2, 3, and 4 all satisfy the criterion. Implementation 2 has a Nil value for Subsystem which implies a Nil value for the String$Equal operation. The union operation, however, keeps the Nil value's effect local to one subcondition. Since the other subcondition's value is True the implementation satisfies the criterion.

5.8 Example

This section presents an example selecting an implementation for a set abstraction. Figure 5-8 depicts a set of twelve implementations in the abstraction's du. The implementations are partitioned into three alternatives based on data representation: list, tree, and characteristic vectors. In the vector alternative, Implementations 4 through 6 trace debugging history; Implementations 9 and 10 do the same for the tree alternative. The list alternative, however, has a more complex structure. Implementations 1 through 3, 11, and 12 represent sets as unordered lists. Implementations 11 and 12 check for and eliminate duplicates at insertions; Implementations 1 through 3 do not. Sorted lists are used in Implementations 7 and 8.

The figure also shows some of the attributes applied to these implementations. CreatedBy, Alternative, DefaultForAlternative, and DefaultForDU are mandatory standard attributes defined in Section 4.5. Project, Subsystem are string-valued optional standard attributes. LanguageSubset is a customized attribute bound to a type of the same name. DeletionWork give a gross measure of how difficult it is to do a deletion in an given

Figure 5-8:Retrieval in a Set Abstraction



implementation. Its type, Measure, is an enumeration type with values *Easy*, *Medium*, and *Hard*.

Figures 5-9 to 5-15 trace the selection of an implementation for the Set abstraction du using the following characterization:

Requirements: •.Project = "XYZ" & •.Subsystem = "ABC"
∪
•.CreatedBy = User # Bill

Preferences: •.Alternative = "List"
•.LanguageSubset = LanguageSubset # NoAnys
•.DeletionWork = Measure$Minimize(••.DeletionWork)

Minimize is a function that yields the smallest attribute value for DeletionWork found on any of the implementations in the criterion's input set. The Library options for the automatic filters for supersession and the defaults filters are set. For the sake of readability, the figures show only those attributes used in subsequent criteria.

The requirements criterion eliminates all the implementations in the vector alternative plus implementation 3 in the list alternative because they satisfy neither of the subconditions joined by the union operator. Implementations 9 and 10 meet the Project and Subsystem subcondition; Implementations 1, 2, 7, 8, 11, and 12 meet the other. The union operator acts like a set union operator, merging the results of these two subconditions. to yield the implementations shown in Figure 5-9 as the criterion's result set.

The first preference criterion, taking Figure 5-9 as its input set, eliminates the tree alternative from consideration. Its result set, Implementations 1, 2, 7, 8, 11, and 12, is shown in Figure 5-10.

No implementations satisfy the second preference criterion. Of the implementations in its input set, only Implementations 7 and 8 have the attribute LanguageSubset. Neither of

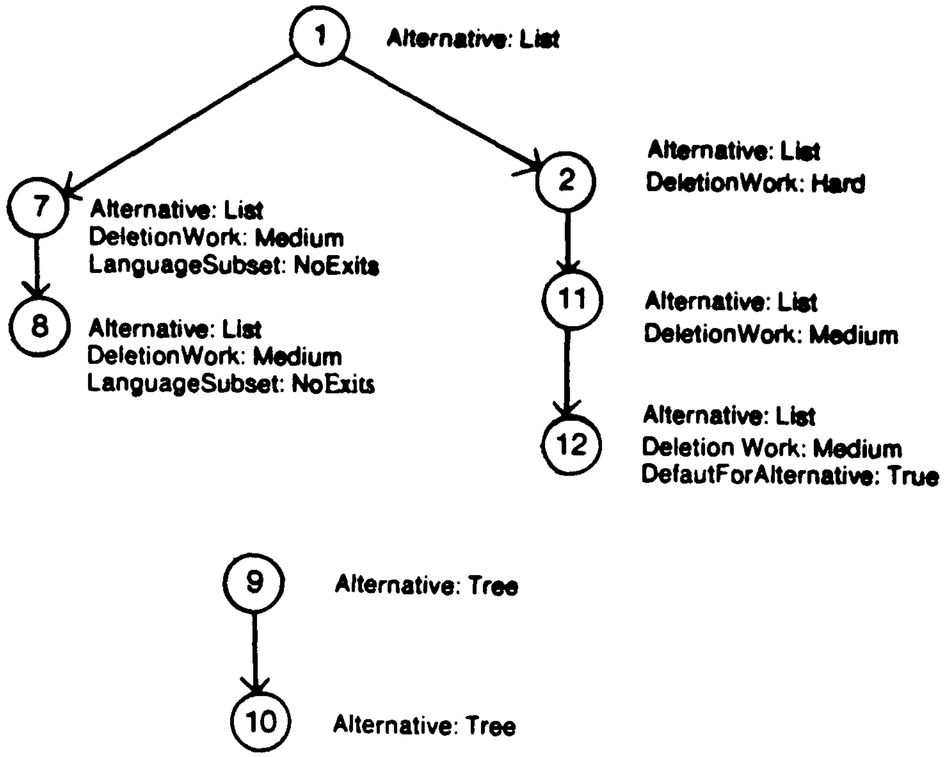


Figure 5-9:Retrieval - Result Set of the Requirements Criterion

those have the correct value, so the expression's value is False. For all the others, the expression yields Nil. The criterion is void and its input set is passed on as its result set (Figure 5-11).

The invocation of Measure$Minimize does not depend on the current implementation. It is thus subject to optimization. Before considering any implementation, the Library computes the value of the Measure$Minimize invocation. When evaluating the criterion expression for each implementation in the input set, the previously computed value is substituted for the invocation.

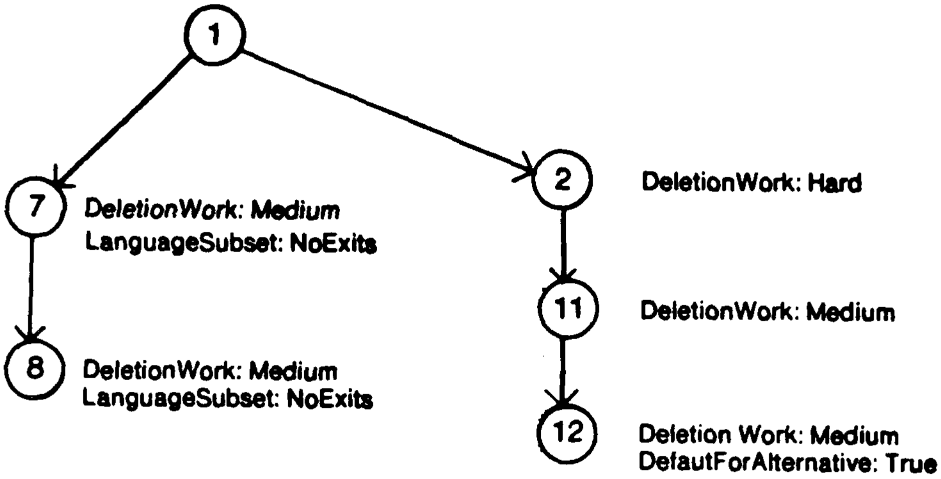


Figure 5-10:Retrieval - Result Set of the First Preference Criterion

For implementation 1 the expression yields Nil because Implementation 1 is missing this attribute. The minimum value, as found by Minimize, is Medium, so Implementations 7, 8, 11, and 12, are selected (Figure 5-12).

The Supersession filter eliminates Implementations 7 and 11 because they are superseded by 8 and 12, respectively (Figure 5-13). The DefaultForDU filter does not apply because the implementation with DefaultForDU set to True, Implementation 6, is not contained in its input set. Its result set, shown in Figure 5-14, is therefore equivalent to its input set.

Finally, the DefaultForAlternative filter yields a single implementation result (Figure 5-15). Both implementations in the input set are part of the list alternative and

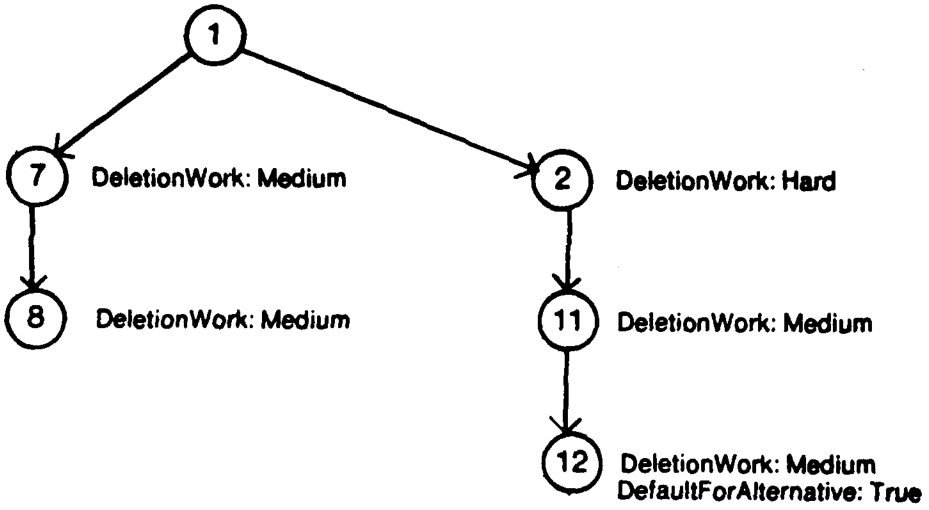


Figure 5-11: Retrieval - Result Set of the Second Preference Criterion

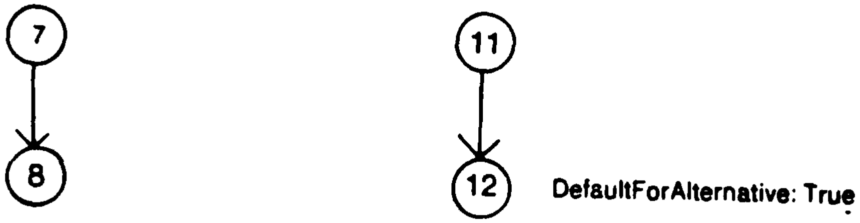


Figure 5-12: Retrieval - Result Set of the Third Preference Criterion

---

Implementation 12 has the value True for *DefaultForAlternative*. Implementation 12 is the result set of the whole selection process and is retrieved.

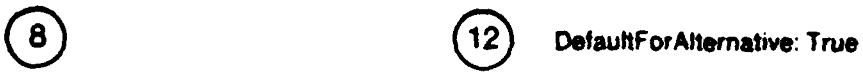


Figure 5-13: Retrieval - Result Set of the Supersession Filter

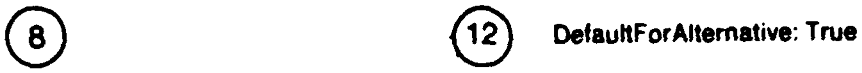


Figure 5-14: Retrieval - Result Set of the Du Default Filter

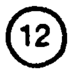


Figure 5-15: Retrieval - Result Set of the Alternative Default Filter

Chapter Six

Program Composition

Chapter 5 explored the use of the attribute naming scheme in retrieving an implementation of an abstraction from a du in the CLU Library. A more complex implementation retrieval operation occurs in program composition, the process of assembling a set of implementations into a working program called a composition.

In this chapter we explore a descriptive or non-procedural approach to program composition for use in the CLU programming support environment. The user identifies an abstraction for which a composition is to be built, and specifies the properties he would like the composition to have. As with selecting an implementation for a single abstraction, properties are represented by conditions on attributes. The Library determines what abstractions are contained in the composition and selects an implementation satisfying the specified properties for each.

The Library Compose command embodies composition based on the properties of programs and their constituent abstractions. It has the form

Compose <Composition-characterization>

Executed in the context of the du representing the top-level abstraction of the program, Compose builds a composition for the abstraction and stores it in the abstraction's du. The <Composition-characterization>, or characterization for short, describes the properties of the composition built, and is used to determine which implementations of each of the component abstractions to include. We say that the composition built *satisfies* the characterization.

Using the Compose command as a basis, this chapter explores the descriptive approach to program composition. Composition is a complex problem, offering many challenges that we cannot hope to solve in this document. It is not our intention to propose a finely-tuned solution. Instead, we analyze the issues involved, presenting a simple solution only as an aid to the analysis.

The chapter is organized as follows. An analysis of the issues raised by composition occupies Section 6.1. Section 6.2 presents a simple proposal for Compose. Examples and discussion of the proposal are contained in Section 6.3. Section 6.4 considers the problem of modifying an already existing composition. Finally, Section 6.5 summarizes our findings about composition.

6.1 An Analysis of Program Composition

Program composition consists of two subproblems: identifying the abstractions comprising a program and selecting implementations for each. As will be shown below, in the CLU programming support environment the first can be reduced to naming the top-level abstraction of the composition by using information stored in the Library. The second subproblem therefore comprises the heart of the problem.

6.1.1 Identifying the Component Abstractions

The abstractions comprising a program are the top-level abstraction, its subsidiary abstractions, their subsidiaries, and so on. By invoking the Compose command in the context of its du, the user identifies the top-level abstraction. The other abstractions can be deduced from information in the Library.

Because two implementations of an abstraction may depend on different abstractions, the subsidiary abstractions of an abstraction depend on the choice of implementation. As discussed in Chapter 2, the names of the abstractions referred to in the CLU source module of an implementation are bound to dus at compile-time. This information is saved in the Library and provides explicit identification of subsidiary abstractions.

Thus, the Library can iteratively identify the abstractions comprising a program as it selects implementations for the abstractions it already knows. Given the identity of the top-level abstraction, the first step is to select an implementation for it. The Library can then look up the dus representing the implementation's subsidiary abstractions in the Library, and select implementations for them. It can look up those implementations' subsidiaries, and so on.

Interleaving implementation selection with abstraction identification, therefore, reduces the problem of identifying a program's constituent abstractions to naming the top-level abstraction. To determine the rest of the abstractions, Compose uses the information in the Library to walk the dependency tree of the composition being built. Nested selection operations choose implementations for each of the abstractions found in the dependency tree.

6.1.2 Selecting Implementations

Chapter 4 analyzed implementation selection for a single abstraction. Implementation selection in the context of composition is a more complex operation involving multiple abstractions. This section analyzes this compound form to see how it differs from the single abstraction case.

One difference between retrieval from a single du and retrieval for composition is that Compose must select only compiled implementations. Because its function is to build an executable program, Compose cannot use an uncompiled version. Compose can retrieve a compiled version either by considering only those implementations that have already been compiled, or by considering all existing implementations, and invoking the compiler if an uncompiled implementation is selected. Although either approach is feasible, for simplicity in this work we choose the former.

We would like Compose to parallel Select as much as is feasible, drawing on ideas developed for selection in a single du. In particular, we would like to have Select and Compose be compatible, sharing mechanisms and semantics as much as possible. Subject to the restriction requiring selection of compiled implementations only, we would like every implementation characterization accepted by Select to be a valid composition characterization for Compose, and to have all characterizations accepted by both be treated identically by the two operations.

We believe that implementation selection for a single abstraction and for composition are very similar. For the single du case of Chapter 5, we modeled descriptions of implementations as sets of properties and introduced issues such as weighting of properties, default application of common properties, and the treatment of properties involving unspecified values. The model and all the issues raised in Chapter 4 are relevant to implementation selection as a part of composition. However, we identify two issues introduced by composition that are not present in the single abstraction case. The following sections discuss them in detail.

6.1.2.1 Partitioning Users' Knowledge

To select an implementation of an abstraction, the user must understand what implementations are available for use. But in a large program composed of many abstractions a user often has varying levels of familiarity with different parts of the program.

Some abstractions he may understand very well, others only slightly. For example, each user is typically very familiar with the abstractions he is responsible for and less familiar with the abstractions written by colleagues responsible for other parts of the program. For an abstraction not created specifically for the user's project but instead borrowed from the store of general-purpose abstractions available in the Library, the user is likely to have almost no information at all.

Because the user's understanding of the abstractions in his program is so varied, he may want to describe distinct parts of the program at different levels of detail, corresponding to his levels of understanding. He could describe the abstractions he maintains using very detailed descriptions of functional properties and perhaps even du-specific properties. Knowledge of program evolution might also be important. For abstractions created by other members of the project, he might use properties based on configuration management information and would depend less on program evolution. For general-purpose abstractions found in the Library, no implementation is tailored especially for his project. Hence, the user would probably not attempt any description at all, relying instead on the Library defaults to select the designated version.

Our conclusion is that a single description is inadequate. Because a user's understanding of the abstractions in a program varies, he might want to partition the description correspondingly. Partitioning the description allows him to express the properties of the program in the way he feels is most natural, providing the maximum amount of detail for those parts of the program he understands well and less for the others. Without a partitioning mechanism, he would be forced to maintain a uniform level of description across all abstractions in the system, either omitting detail for the abstractions he understands well or fabricating detail for the abstractions with which he is less familiar.

6.1.2.2 The Library as a Non-Uniform Database

Not all attributes are relevant to all abstractions. Different attributes make sense in different dus. For example, some attributes, like CreationTimestamp, are applicable to all abstractions; others might be relevant only to a limited set of abstractions, such as the abstractions developed by a single user, or comprising a particular program. Du-specific attributes are the extreme case of this, applying only to a single abstraction.

We say that an attribute that is relevant to an abstraction is *meaningful* for the abstraction and its du. An attribute that is meaningless has no reasonable values in the abstraction's du. We say that its value for each implementation in the du is *undefined*.

Attribute meaningfulness impacts program composition because the composition characterization describes properties of the desired system in terms of attribute values. A characterization referring to an attribute not meaningful for some du is not interpretable with respect to the du because it specifies a property that doesn't make sense. In composition, if the characterization refers to an attribute not meaningful across all abstractions in the program, then the Library will not be able to interpret the characterization and select implementations for all of the abstractions in the program.

One solution would prohibit the use of meaningless attributes in characterizations: composition fails if the Library encounters a reference to an attribute not meaningful for some abstraction in the program. We reject this solution for three reasons. First, it is inflexible, constraining the set of attributes that can be used in composition to those meaningful across all abstractions in the program. This approach discourages the user from using du-specific attributes or attributes relevant to only a subset of the abstractions in a program. To use such an attribute, the user is forced to manufacture nonsensical values for the attribute in abstractions where it is meaningless. Second, for every new abstraction added to a program, the user must supply values for all the attributes used in composition of the containing program. This could require significant effort, deterring the user from modifying his program. It also conflicts with attribute independence, one of the six characteristics of naming systems supporting implementation identification. Finally, it discourages the user from borrowing existing abstractions from the Library because they would not have program or project specific attributes defined.

Therefore, we need a way to interpret the characterization and select an implementation from a du in the presence of meaningless attribute references. An intuitive approach is to treat an undefined attribute like the unspecified attribute value discussed in Chapter 4. Selection in the presence of meaningless attribute references is accomplished by interpreting those references identically to references to unspecified attribute values. This approach is reasonable because, although undefined refers to attributes and abstractions while unspecified refers to attributes and implementations, both represent unknown attribute values. Properties based on either have unknown truth values and can be viewed as the same.

However, one can easily imagine situations in which the difference between undefined and unspecified values is significant. For example, consider a property based on an attribute Project, recording the name of the project to which each implementation belongs. Abstractions maintained in the Library as part of the general store available for use by all are not created for a specific project; the Project attribute is meaningless in their dus. Implementations created specifically for a user's project are identified using this attribute. In selecting an implementation from a du, the user wants to ignore properties based on the Project attribute if it is not meaningful because those abstractions come from the general store and are not project-specific. If the Project attribute is meaningful, the user wants to select only from those implementations explicitly identified as belonging to his project. In general, a user might want a property to matter only if it makes sense. That is, a property referring only to meaningful attributes is satisfied only by implementations with explicitly specified values; all implementations satisfy properties referring to meaningless attributes.

An interesting problem concerns how to determine whether or not an attribute is meaningful for some abstraction. Clearly, if any of the du's implementations have the attribute applied, then it is meaningful. But suppose an attribute is not applied to any of the implementations. Is there any way to know whether it is meaningful or not? Perhaps it is meaningful, but none of the values for the implementation have been specified (and all are therefore Nil). We claim that only the user can know what attributes are meaningful in the context of a du. Any proposal for composition must also include a mechanism allowing the user to declare which attributes are meaningful in a du.

6.2 A Simple Proposal for Compose

This section presents a simple proposal for Compose. Compose walks the dependency tree of the program, selecting an implementation for each of its component abstractions. Interleaving abstraction identification with implementation selection, as described in Section 6.1.1, allows Compose to use the information about subsidiary abstractions stored in the Library to identify the abstractions comprising the program.

For each of the abstractions encountered in a program's dependency tree, a compiled implementation is selected based on the composition characterization and on Library options. Selection in the context of composition closely parallels the Select command, using a composite characterization split into discrete conditions called criteria. Each criterion is used as a filter on the set of implementations, resulting in a semantics of repeated subsetting as each criterion filter is applied to the result of the previous criterion.

A single mechanism, based on a more complex form of criterion, addresses the two issues (discussed in Section 6.1.2) distinguishing implementation selection for composition from the single-du case. Each criterion may have an associated *applicability test* that controls whether or not the criterion is applied in a given du. The test is evaluated once for each du in the dependency tree. Only those criteria whose applicability tests evaluate to True are applied in a given du.

Applicability tests directly support a user's ability to partition a characterization to accommodate his differing levels of knowledge. The effect of each criterion may be isolated to a subset of a program's abstractions by imposing an applicability test True only for those abstractions.

Applicability tests also allow a user to control the interpretation of undefined attributes. We use a single unknown value to represent both undefined attributes and unspecified attribute values. Consequently, the default case views them as identical. To treat undefined attributes differently from unspecified attribute values, a user supplies multiple criteria with applicability tests that check for attribute meaningfulness.

6.2.1 Walking the Dependency Tree

Compose uses information stored in the Library recording each compiled implementation's subsidiary abstractions to simultaneously construct and walk a program's dependency tree. Compose maintains a list of dus representing abstractions in the program for which implementations may not yet have been selected. For each du in the list, it selects a compiled implementation, looks up the dus representing the implementation's subsidiary abstractions in the Library, and adds them to the list. As its implementation is selected, the du is eliminated from the list. Initially, the list contains only the du representing the top-level abstraction, given by the user as the context for the Compose command The Library selects an implementation for this abstraction and adds the the dus for its subsidiaries to the list. As their implementations are selected, their subsidiaries are added, and so on, iteratively selecting implementations for all abstractions in the dependency tree.

Often an abstraction will be referenced by several abstractions in a program, causing it to be added to the list repeatedly, as a subsidiary of each of the referencing abstractions. Compose does not repeat selection each time. Instead, it checks to see if an implementation has already been selected for a du, before proceeding with implementation selection.

Compose fails if an implementation cannot be selected for one of the abstractions in the program. This occurs either because no implementations exist or because implementations exist but none are acceptable to the user. If implementations exist, but the normal selection process yields the empty set, the user is offered the opportunity to select one himself and prevent the failure.

6.2.2 The Characterization

Like the implementation characterization of the Select command, the composition characterization is an ordered set of subconditions called criteria. The set of criteria is partitioned into requirements and preferences. Although the implementation characterization of Select contains only a single requirements criterion, the composition characterization might have an arbitrary number of requirements criteria as well as an arbitrary number of preference criteria.

Criteria for composition take a more complex form than in implementation characterizations. The following grammar describes the legal criteria. Non-terminals appear inside angle brackets. The vertical bar denotes or.

```
<criterion> ::= <simple-criterion> | <qualified-criterion>
<qualified-criterion> ::= <applicability-test> ⇒ <simple-criterion>
<simple-criterion> ::= <expression>
<applicability test> ::= <expression>
```

Each criterion is either a simple criterion or a qualified criterion. Identical to the criteria used in implementation characterizations, a simple criterion is an expression that evaluates to a three-phase logic value. A qualified criterion consists of a nested simple criterion and an applicability test used to control whether or not the nested criterion is applied. We say the test *qualifies* the criterion. Both the applicability test and the nested criterion are arbitrary three-phase logic expressions.

6.2.3 The Semantics of Implementation Selection

Patterned after implementation selection for Select, Compose applies a series of filters to the set of implementations contained in a du, successively subsetting the set of implementations under consideration. Compose first applies the requirements criteria as filters, followed by the preferences criteria. Then, under user control via Library options, it automatically applies the three filters based on y attributes described in Section 5.4: the Supersession filter, the DefaultForDU filter, and the DefaultForAlternative filter. The user is consulted if this selection process does not yield a unique implementation.

The two kinds of criteria are evaluated differently. A simple criterion, consisting of a single three-phase logic expression, is analogous to the criteria of Select. It is evaluated once for each implementation in the criterion's input set, with the symbols * and ** bound to the current implementation (the implementation being evaluated) and the current input set (the criterion's input set), respectively. Attribute values are read using ReadAttribute and ReadAttributeOnSet. The result set, containing those implementations satisfying the criterion, is the subset of implementations in the input set for which the expression evaluates to True. For the sake of efficiency, subexpressions of the criterion expression not dependent on the current implementation are evaluated once instead of repeatedly.

A qualified criterion contains two three-phase logic expressions, an applicability test and a nested simple criterion. A qualified criterion is evaluated as follows. First, the applicability test is evaluated. If the test yields True, then the nested simple criterion is evaluated once for each implementation in the current input set, as described above, with the result set containing those implementations for which the nested criterion evaluates to True. If the applicability test does not yield True, the qualified criterion's result set is the same as its input set. In both expressions, attribute values are read using ReadAttribute and ReadAttributeOnSet, and the symbol ** is bound to the criterion's input set. The symbol * has no meaning in the applicability test but refers to the current implementation in the nested simple criterion.

Selection fails and the user is consulted if the result set of a requirements criterion is empty. If the result set of a preference criterion is empty, its input set is passed along as the result set.

6.2.4 Unknown-Valued Attributes

We use the single unknown value Nil, defined in Chapter 4, to represent both undefined and unspecified attribute values. Thus, the attribute reading functions ReadAttribute and ReadAttributeOnSet return Nil for both meaningless attributes and meaningful but unspecified attributes. All functions except the three-phase logic operators, ∪, ∩, and ~, preserve Nil.c4

Two Library commands allow a user to control what attributes are meaningful in a given du.

Meaningful <Attributename>

is executed in the context of a du and tells the Library that the named attribute is meaningful in the du.

Meaningless <attributename>

is the opposite, telling the Library that an attribute previously considered meaningful in the du is now meaningless. An attribute name for which neither command has been invoked is assumed to be meaningless. We assume that any attribute applied to an implementation in a du is meaningful for the abstraction, so applying an attribute to some implementation automatically invokes the Meaningful command. *The reverse, however, is not true.* Removing the last use of an attribute from a du does not necessarily imply that the attribute is no longer meaningful for the abstraction. Therefore, we make the user explicitly invoke the Meaningless command to make an already known attribute meaningless for a du.

The Library maintains with each du a list of the attributes that are meaningful. The user can access this information using a function provided by the Library.

IsMeaningful (Attributename)

when called in the context of a du returns True if the attribute is meaningful in the du and False otherwise.

6.2.5 Summary of Composition Semantics

This section presents pseudo-code summarizing the above proposal. The pseudo-code is divided into three parts. The first is the top-level of Compose and gives the algorithm for walking the dependency tree of the composition being built. The second and third parts are supporting procedures. One selects an implementation from a du given a characterization; the other filters a set of implementations using a given criterion.

```
Let C be the characterization supplied by the user
Let S be the set of dus for which implementations have not yet been selected
Let T be the set of dus for which implementations have already been selected
Let U be the set of selected implementations

S := the singleton set consisting of the du for the top-level abstraction
T := the empty set
U := the empty set

For each du D in S do
    If D ∈ T then delete D from S
    Else select an implementation I from D by invoking SelectImplementation (D, C)
        Insert I into U
        Move D from S into T
        Look up the dus representing subsidiary abstractions of I
            and insert them into S
End For
Build a composition from the implementations in U and store it in
    the du for the top-level abstraction.
```

S contains dus in the dependency tree for which implementations might not have been found; T contains those for which implementations have already been retrieved. Initially, T is empty and S contains only the root du. As each implementation is selected, its du is transferred from S to T and the dus for its subsidiary abstractions are added to S. Retrieved implementations are saved in the set U. The set T is checked before selecting an implementation for a du in case one has already been selected.

```
SelectImplementation = Procedure (D: du, C: characterization)
    Returns (I: Implementation)
Let RS be the empty result set
Let IS be in the set of implementations in the du D

% Apply each criterion as a filter
For each criterion B do
    RS := Filter (IS, B)
    If RS is empty then
        If B required then selection fails - Consult the user
        Else RS := IS
    IS := RS
End For

% Apply the automatic filters
If the Library Supersession option is set then
    RS := result set of Supersession filter applied to IS
    IS := RS
```

```
If the Library DefaultForDU option is set then
    RS := result set of DefaultForDU filter applied to IS
    IS := RS
If the Library DefaultForAlternative option is set then
    RS := result set of DefaultForAlternative filter applied to IS

% Test for Non-unique result -- RS contains the result of selection
If $o(RS) \neq 1$ then consult the user
Else return the implementation in RS
```

Given a du and a characterization, this function tries to select an implementation satisfying the characterization. It first applies the characterization's criteria as filters, then applies the automatic filters based on supersession and defaults. If a requirements criterion filter yields the empty set, or if the selection process does not produce a single implementation result, the user is consulted.

```
Filter = procedure (IS: the criterion's input set of implementations. B: Criterion)
          returns (RS: result set)

% Qualified Criterion
If B is a qualified filter with applicability test A and nested simple criterion G then
    Evaluate A with IS bound to **
    If A is True then RS := filter (IS, G)
    Else RS := IS

% Simple Criterion
Else RS := the empty result set
    For each implementation I in IS do
        Evaluate B with IS bound to ** and I bound to *
        If B is True then insert I into RS
    End For
Return (RS)
End Filter
```

This procedure filters a set of implementations using a given criterion. For a qualified criterion it evaluates the applicability test, recursing to filter the set using the nested simple criterion only if the test yields True. A simple criterion is evaluated once for each implementation in the set, selecting those which yield True.

6.3 Examples and Discussion

In this section we demonstrate how our proposal addresses the two issues introduced by program composition. We then present an example using our proposal to compose a program consisting of four abstractions.

6.3.1 Sample Applicability Tests

The first of the two issues not present in retrieval for a single du concerns the partitioning of a user's knowledge. Applicability tests can be used to partition a characterization as follows. A simple criterion is applied to all abstractions in a program, but the nested simple criterion of a qualified criterion is applied only to those abstractions in which the applicability test evaluates to True. Thus, an applicability test can limit the effect of a criterion, identifying the abstractions to which the nested criterion is to be applied by checking for attributes and specific attribute values.

The following series of sample applicability tests demonstrate this ability to partition a characterization. ••.<attribute name>, an invocation of ReadAttributeOnSet, denotes a mapping from implementations to attribute values, and • <attribute> invokes ReadAttribute to read the value of an attribute on a single implementation. Mapping[T]$Isin checks for a specific value in a mapping from implementations to values of type T. The # notation again denotes a user-defined type literal.

First, three applicability tests can be used to partition the abstractions in a program into three classes depending on how they were developed.

Mapping[User]$Isin (User # Me, ••.CreatedBy)

¬Mapping[User]$Isin (User # Me, ••.CreatedBy) &
  Mapping[String]$Isin ("Mine", ••.Project)

¬Mapping[String]$Isin ("Mine", ••.Project)

CreatedBy is a mandatory attribute of type User. Project is an optional attribute bound to strings The first of the three applicability tests identifies dus for abstractions containing implementations developed by the user named "Me". If "Me" refers to the user composing the program, then criteria associated with this test are only applied in dus representing abstractions for which the user is responsible. The second example identifies abstractions created by other members of the user's project The third identifies abstractions not created specifically for the user's project.

Two other examples partition a characterization based on optimization needs and history. The applicability test

IsMeaningful (OptimizedFor)

checks to see if a customized attribute OptimizedFor recording optimization requirements is meaningful. It is associated with criteria to be applied only for abstractions in which optimization strategies are important. Finally,

Date$After (Date # 6/18/83, ••.CreationTimestamp)

uses values of the mandatory attribute CreationTimestamp to identify recently modified abstractions.

The second issue affecting retrieval in the context of composition is the possibility of undefined attributes. The analysis in Section 6.1.2.2 found that two conflicting interpretations of meaningless attributes are possible. One view unites undefined attributes with the unspecified attributes discussed in Chapter 4; the other makes a distinction between them. Both are reasonable. Using applicability tests and a single unknown value, our proposal allows the user to control which interpretation is taken.

Our proposal assumes the first view as the default case, representing both undefined and unspecified attributes by a single unknown value, Nil. To force the second interpretation, the user writes one or more qualified criteria, testing for attribute meaningfulness in the applicability test. For example, simple criterion

```
    *.Project = "Mine"
```

unifies the two unknown values and takes the first view. Qualified criteria distinguish between them. The user can write the

```
    IsMeaningful (Project) ⇒ *.Project = "Mine"
```

to skip the criterion if Project is undefined. Or he may write a sequence of criteria,

```
    IsMeaningful (Project) ⇒ *.Project = "Mine",
    ¬IsMeaningful (Project) ⇒ *.CreatedBy = User # Me
```

to substitute a property based on the CreatedBy attribute if Project is not meaningful.

6.3.2 A Composition Example

In this section we offer an example of composition for a program consisting of four abstractions. We will first look at implementation selection for the top-level abstraction of the program, then trace the abstraction identification process as Compose walks the program's dependency tree. We omit implementation selection for the three non-root abstractions because of the similarity to selection for the top-level abstraction.

The user invokes Compose in the context of the du for abstraction A using the following composition characterization

```
Requirements:   *.Project = "X",
                Mapping[User]$Isin (User # Bill, **.CreatedBy) &
                        IsMeaningful (OptimizedFor)  ⇒
                        *.OptimizedFor = ProjectOptimization # Retrieval
Preferences:    *.Datasize = Size # Small,
                Mapping[String]$Isin ("Y", **.Subsystem)  ⇒
                        *.Subsystem = "Y" & *.ReleaseState = Release # Public
```

where Project and Subsystem are string-valued attributes, OptimizedFor and ReleaseState are attributes having user defined types, and CreatedBy is a mandatory attribute of type User.

Figure 6-1 displays the set of implementations for abstraction A. We assume that in addition to the y attributes only those attributes displayed in the figure are meaningful in the du. We also assume that the Library options are set to allow the filters based on supersession and defaults to be automatically applied.

To build a composition for A, Compose's first step is to select an implementation from A's du, successively applying the requirements criteria, the preference criteria and the automatic filters The first criterion applied specifies that the implementation be associated with Project X. Implementation 1 has a Nil value for the attribute Project and is eliminated, all others satisfy this criterion.

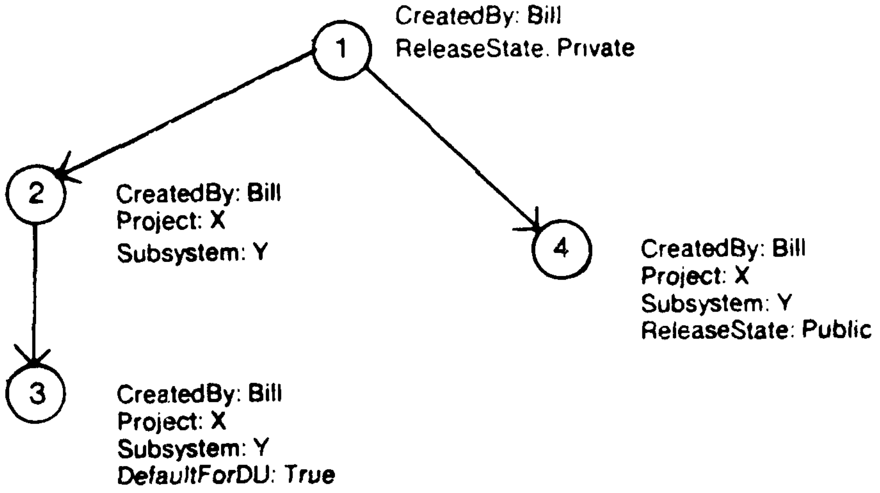


Figure 6-1:A Composition Example. The Top-Level Du

The second requirements criterion is a qualified criterion, so the applicability test is evaluated before examining any of the implementations. Because the attribute OptimizedFor is not meaningful in this du. the applicability test fails The nested simple criterion is not applied. The result set of the criterion is the same as its input set (Implementations 2, 3, and 4).

Next, the first preference criterion is evaluated The value of Datasize for every implementation is Nil because Datasize is not meaningful in this du Thus. none of the implementations satisfy this property, and the result set of the filter is empty As a preference criterion, however, the criterion's input set (Implementations 2, 3, and 4) is substituted for its empty result set.

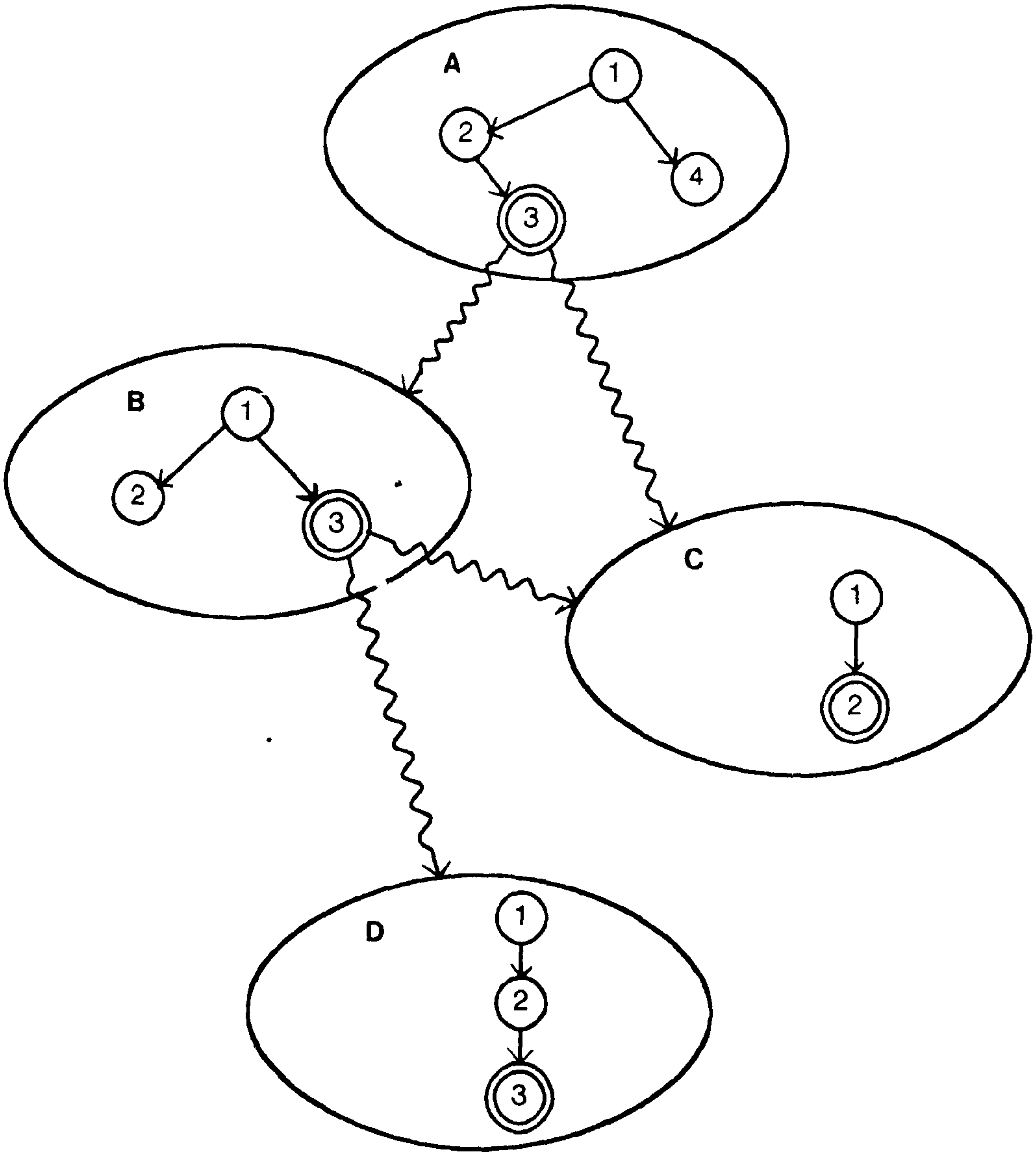
The last preference criterion is qualified by a condition on the Subsystem attribute. Several of the implementations are associated with Subsystem Y, so the applicability test succeeds, allowing Compose to evaluate the implementations in the criterion's input set. Implementation 2 does not satisfy the criterion because it has the value Nil for attribute ReleaseState. The result set of the criterion contains Implementations 3 and 4.

Finally the automatic filters based on mandatory attributes are applied. The Supersession filter does not narrow down the set of implementations under consideration, but the DefaultForDU filter selects Implementation 3.

Figure 6-2 shows the dependency tree of the program rooted at A. As always, small numbered circles represent implementations and straight arrows mark supersession relationships. Large ovals represent dus. Wavy arrows from an implementation to a du represent references by the implementation to a subsidiary abstraction.

Compose now looks in the Library to see what abstractions are referenced in the selected implementation of A. It finds that abstractions B and C are the implementation's subsidiary abstractions. Turning to C, it selects Implementation 2. Implementation 2 has no

Figure 6-2: A Composition Example: The Dependency Graph



subsidiary abstractions. For B it selects Implementation 3, whose subsidiaries are abstractions C and D. Note that C is the subsidiary of both A's and B's implementations. Before selecting an implementation for C as a subsidiary of B, it checks to see if an implementation has already been selected. Because one already has, reselection is avoided. Finally, Compose selects Implementation 3 for abstraction D. The selected implementations are marked with double circles on Figure 6-2.

6.4 Modifying Existing Compositions

Thus far in the chapter, we have considered the process of constructing a composition from scratch, using only the identity of the composition's top-level abstraction and a characterization describing the desired behavior. Alternative methods could use existing compositions either as components or as the bases of new compositions.

One possibility incorporates existing compositions of subsidiary abstractions of a program as components of the composition being built. The names of implementations can change, however, and the same characterization used twice may retrieve different implementations or build different compositions. Consequently, we do not understand when it is safe to use an existing composition as a component of another. This problem is a manifestation of a larger issue. We defer its discussion to Section 7.3.

In this section, we examine the problem of modifying an existing composition. As defined in Section 2.1, a composition is a set of implementations that can be combined into an executable program. Modifying a composition means replacing an implementation with another implementation of the same abstraction.

Every implementation included in a composition except the implementation of the program's top-level abstraction implements a subsidiary abstraction of some other implementation in the program. A composition must include an implementation of the top-level abstraction and implementations of all the abstractions needed by other implementations in the composition. Consequently, a single implementation cannot be eliminated or replaced with an implementation of another abstraction.

Replacing one implementation, however, may force other changes to the composition. Because the new implementation may have different subsidiary abstractions than the old one, we view modification of a composition as an operation on a subgraph of the composition's dependency graph rooted at the implementation being changed.

An example shows how this is true. Figure 6-3 depicts the dependency tree of a composition of abstraction A. Each node represents an implementation of an abstraction; the abstraction is identified by the letter marking the node. Wavy directed arrows indicate dependencies: the implementation at the tail of the arrow depends on the abstraction implemented by the node at the head. The implementation of B depends on abstractions D and E. Modifying the composition of A by replacing this implementation of B with an implementation depending instead on abstractions D and F is an operation on the subgraph rooted at the node implementing B. The implementation of E is eliminated from the composition and an implementation of F must be added. In addition, under different circumstances, the user may or may not want to replace the implementation of D.

We propose an alternative composition command that modifies an existing composition, replacing implementations and possibly abstractions. The Library Recompose command

Recompose <Composition> <Implementation> <Composition-characterization>

embodies this idea. <Composition> is an expression denoting a composition to be modified.

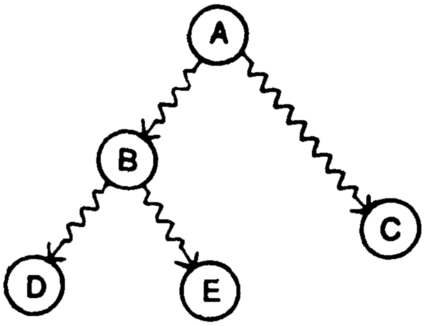


Figure 6-3: The Dependency Graph of a Composition

<Implementation> is an expression denoting the implementation that acts as the root of the subgraph to be changed. The implementation must be contained in the composition. The composition characterization is used to select new implementations for the abstractions in the subgraph being modified. A *Library* option allows the user to control whether or not new implementations are selected for abstractions (such as D, above) in the modified subgraph for which the composition already contains implementations.

This simple recomposition operation address the two kinds of changes that we expect would be most common. One change replaces an implementation containing a bug with another implementation of the same abstraction. The user specifies this type of change simply by replacing the buggy implementation. Another common change occurs when a user incorporates a private version of a subsystem into a standard version of a program. In this case, the user replaces the implementation of the top-level abstraction of the subsystem. If other patterns of modification are desired, they can be achieved by identifying the nodes in the dependency graph to be changed and iteratively replacing their subgraphs.

6.5 Summary and Evaluation

In our analysis of program composition, we found that composition can be divided into two subproblems, abstraction identification and implementation selection. Using information in the Library about subsidiary abstractions, the former can be reduced to identifying the top-level abstraction of a program, leaving the latter as the interesting part of the problem. Drawing on our understanding of implementation selection in a single du, we found that implementation selection in the context of composition introduces two issues not present in the single du case. Our goal was to develop a proposal for composition that would address these two issues while preserving as much parallelism with single du selection as possible.

Section 6.2 presents a proposal for Compose that meets this goal. It shares mechanisms and semantics with the single du case. Selection in Compose is an extension of Select: every implementation-characterization is a legal composition-characterization and analogous characterizations are treated identically by the two commands, subject to the constraint that Compose retrieves only compiled implementations. A variation on Compose presented in Section 6.4 allows a user to create a new composition by modifying an old one.

Both Compose and Recompose address the two issues distinguishing composition from retrieval of an individual implementation. The use of qualified criteria allows users to partition a characterization, using different criteria to describe distinct abstractions or sets of abstractions contained in a program. Qualified criteria testing attribute meaningfulness allow users to control the interpretation of undefined attributes.

Our simple proposal is a rough first cut at a solution to the problem of program composition. It is intended only as a mechanism for exploring the issues. Although it is adequate for simple cases and has sufficient expressive power for complex cases, it can be tedious to write and therefore inconvenient to use.

For example, in our solution, applicability tests are associated with individual criteria. When an applicability test is shared by several criteria, it must be repeated in the characterization for each of the sharing criteria. Thus, the user must specify

$$T \Rightarrow C_1, \\ T \Rightarrow C_2$$

for two criteria, C_1 and C_2, sharing a common test T. Similarly, two qualified criteria with complementary applicability tests are needed to specify criteria that are alternatives:

$$T \Rightarrow C_1, \\ \neg T \Rightarrow C_2.$$

We can imagine solutions that would reduce the amount of repetition and make characterizations more convenient to use. For example, a grouping mechanism could allow several criteria to share an applicability test, as in

$$T \Rightarrow \{C_1, C_2\}.$$

However, the interpretation of this construct is not immediately clear. Several questions *must be answered. For example, when and how many times is the applicability test* evaluated? And, how does the grouping mechanism interact with the priority mechanism that is based on the order of criteria? Similar questions arise about possible solutions to the problem of criterion alternatives.

Although we recognize these deficiencies in our proposal, we do not suggest solutions. Further study is needed to develop a more elegant solution.

Chapter Seven

Conclusions

In this chapter we review the work of this thesis. Sections 7.1 and 7.2 first summarize and then evaluate our research. Finally, Section 7.3 suggests directions for further inquiry.

7.1 Summary

This thesis addresses the problem of identifying the implementations of an abstraction. To identify the implementation that best suits his needs, a user must und‧.stand the complete behavior of each of the abstraction's implementations. This is a difficult task because of the sheer volume of information required to describe all the different aspects of many implementations. In addition, when several programmers collaborate on a program composed of many abstractions, they must share their insights about implementations, each teaching the others about the abstractions he is responsible for.

To assist the user with implementation identification, we proposed a system for naming implementations that allows users to capture and record their insights about implementations as names. Names that reflect users' understanding of an implementation's behavior serve as documentation of the implementation's significant features. Expressive names thus help the user to retain and organize the information he needs for implementation identification, and act as a communication medium among users.

We began our work with an analysis of the problem of implementation identification, and determined that six characteristics of naming schemes support the user in this activity.

- The naming scheme should have the capability of expressing a wide range of information in a form meaningful for users.

- Multiple names per implementation should be allowed.

- Non-unique names, referring to more than one implementation, should be supported.

- The naming system should allow users to treat names independently, without considering either other names of the same implementation or the names of other implementations

- The naming system should recognize and highlight commonality among the information conveyed by multiple names.

- Dynamic name creation and modifiability should be supported.

The above criteria suggest goals for designing a naming scheme and provide some axes by which existing naming schemes can be measured.

We founa that a naming system based on attributes satisfies the above criteria and supports implementation identification. Each name in our scheme consists of a pair: an attribute name and a value. An attribute name represents a class of information; its value documents the behavior of the specific implementation with which it is associated. Attribute class definitions specify the range of values an attribute may take. A set of predefined classes of attributes provide a basic vocabulary for describing properties of implementations. These classes express commonly used information and information required by the CLU programming support environment. Users may extend this set by defining new classes to represent information specific to a project or an abstraction. A context mechanism supports the reuse and controlled sharing of attribute names.

The remainder of our work explored mechanisms for identifying implementations exhibiting behaviors specified by the user. A user describes the desired properties by specifying conditions on attributes representing those properties. The Library then retrieves implementations whose attributes satisfy the specified conditions, indicating the desired behaviors are provided.

The first mechanism addresses the problem of selecting an implementation of a single abstraction. Our approach is an extension of the relational database model, addressing issues peculiar to our problem domain. In particular, we introduced the notion of a composite description to express priorities among subconditions, acknowledged a distinction between required conditions and conditions that are merely preferences, recognized the existence of common subconditions, and introduced a three-phase logic system to confront the issue of name independence.

The second mechanism addresses the much more complex problem of program composition. We have shown that selection of implementations for composition is complicated by two issues: first, that users have varied degrees of familiarity with and knowledge about different abstractions comprising a program, and second, that different properties (and therefore different attributes) are relevant to the implementations of distinct abstractions. To explore these issues fully, we proposed a composition scheme based on the mechanism introduced for the single abstraction case.

7.2 Evaluation

The basic issue in evaluating our work is whether or not the naming scheme we propose can help in implementation identification. We believe our work represents a significant improvement over the naming mechanisms commonly used in current practice. However, only experience with our naming system can verify our belief. We have neither built nor used a system of this nature, but we strongly recommend that the next step in investigating naming schemes of this sort be the incorporation of such a scheme into a software database to permit experimentation with its use.

In lieu of experience we can only conjecture. It is our intuition that the effectiveness of our proposal depends on two factors. The first is the interface presented to the user. Our work does not address interface issues. We describe the basic structure of names, present the abstract syntax of Library operations for creating and manipulating names, and explore mechanisms for retrieving implementations from the Library, but do not consider how the names and operations can be integrated into the Library interface. The convenience and user-friendliness of the interface will have a significant effect on the usefulness of our proposal.

A second factor concerns a user's needs when naming implementations. We believe that users often are more interested in a particular behavior than in the specific implementation that exhibits it. In situations where this is the case, a naming system such as ours can significantly aid the user in implementation identification because it offers him the ability to conveniently describe the behavior of implementations. In contrast, current practice in naming implementations relies on unique identifiers which at best are awkward media for encoding descriptions.

For situations in which the user already knows precisely which implementation suits his needs, our proposal is of less assistance. Although in this case unique identifiers are adequate, we maintain that our approach still benefits the user by encouraging planning and coordination in the name space. We encourage users to determine what properties they consider significant, then reflect those properties in implementations' names. Although name spaces of unique identifiers can be similarly planned in advance, users tend to take an ad hoc approach. In addition, as a part of our proposal, we actively encourage users to think about the semantic differences between implementations, expressing these as higher-level names, rather than relying on the primarily syntactic and historical properties most often represented by unique identifiers.

One concern about using the proposed naming scheme is the reliability of information expressed as names. Our approach allows users to document their insights about implementations. But users' insights are not always correct. In contrast, the information derived by tools in the CLU programming environment inspires greater confidence and is more likely to be correct, but tends to be less informative. Use of authorization procedures to safeguard user-supplied information can enhance but not ensure correctness. Users selecting an implementation must be cognizant of the source of the information they use.

We have three comments about the implementation retrieval mechanisms proposed in Chapters 5 and 6. First, many issues combine to make implementation retrieval a very difficult problem. Our goal was to explore implementation identification by suggesting mechanisms to address each of the issues we raised. We believe that our analysis of the issues offers new perspectives on implementation retrieval and that our proposals provide capabilities for property-based retrieval of implementations not previously available.

One drawback of our approach to composition is that it eliminates the need to think about program structure. Approaches based on module-interconnection languages, first proposed by DeRemer and Kron [16], view program structure design as a separate task in program development. Programmers are forced to be consciously aware of the structure of a program. Our approach circumvents this requirement by making the Library responsible for determining program structure during composition. This is a significant loss. Archibald [2] reports that users derive significant benefits from actively considering program structure using techniques such as module-interconnection languages. In response, we suggest investigation into methods by which these two approaches can be combined, preserving the benefits of both.

Another failing is that our program composition mechanism does not supports multiple implementations of an abstraction within a program. It is sometime desirable to use several implementations of an abstraction in the same program, satisfying the differing needs of the abstractions that depend upon them. For example, two subsystems of a program might choose different implementations of a set abstraction if only one of them performs deletions. Multiple implementations of a data abstraction, however, may cause erroneous results. Although multiple implementations of procedural and control abstractions are always safe, the question of when it is safe to have multiple implementations of a data abstraction is not well understood. Consequently, we chose to limit our approach to a single implementation per abstraction as a simplification. Further work is required to extend our proposal to accept multiple implementations per abstraction.

Finally, although our research is based on CLU and the CLU programming methodology, we believe our results have implications for other languages and environments as well. The basis for our work is the distinction between abstractions and their implementations. In CLU this notion is very strong; in other languages the distinction is sometimes weaker. For example, in Ada the distinction is represented by public and private parts of a module. Sometimes, however, the public part contains information that is implementation specific, such as data representation used for compiling procedure invocations in-line. Despite these weaker notions, we believe our approach can be extended to support all languages making a reasonable distinction between abstractions and their implementations. In particular, we believe our approach can be applied in multi-language systems, such as distributed programming environments in which each host machine may choose a different language.

We also rely on the CLU data abstraction mechanism to model the domains of classes of attributes. If an analogous mechanism is not available, domains must be restricted to whatever types are available. As discussed in Section 4.2.2.2, this approach is adequate, although it has several disadvantages.

In CLU, compilation binds an implementation to its subsidiary abstractions. These bindings are used by the program composition mechanism to determine what abstractions comprise a program. For languages in which these bindings are not available as a result of compilation, some other mechanism must be substituted for determining a program's component abstractions. Possibilities include supplying the composition mechanism with a dependency graph or a module-interconnection language description of the program being composed.

7.3 Future Work

The previous section identified engineering and experimentation with an expressive naming scheme as an area in which we believe further research is critical. Engineering entails determining what style of interface will best support the naming system, and implementing the needed mechanisms. Experimentation should provide many insights: identifying commonly used classes of information that should be represented as standard attributes, detecting patterns of attribute use, and so on. It may also identify methodology or policy issues that could support implementation identification. For example, one policy decision might require users to supply attributes to distinguish every existing implementation of an abstraction from every other: no two implementations would be permitted to share an identical set of names.

Further work also is needed in program composition. Our intention was to identify the issues that make composition a hard problem and propose a simplistic solution. Developing a sophisticated solution is an open research problem of significant magnitude. We suggest as one avenue to pursue an attempt to unite our approach with module-interconnection language program structure descriptions. We view program composition as consisting of two subproblems: abstraction identification and implementation selection. Our proposal uses information in the Library recording subsidiary abstractions to accomplish the former. An alternative approach could use a program structure description to determine which abstractions comprise a program. Our selection mechanisms could be used to retrieve an implementation for each of the program's abstractions. Such a solution would offer the benefits of both approaches, requiring users to consciously consider the structure of their programs while retaining the ability to describe implementations and select them on the basis of their behavior

The information expressed as attributes is not secure and cannot be relied upon without an authorization scheme to control creation and modification of names. Although we did not address authorization in our work, a suitable protection mechanism must be designed to support an expressive naming scheme.

As discussed in Chapter 4, the Library can enforce consistency constraints on standard classes of attributes. Consistency constraints support the correct use of attributes by verifying that no two implementations have attributes expressing conflicting information. For example, the Library enforces a constraint prohibiting more than one implementation of an abstraction from having the value True for the mandatory DefaultForDu attribute. This constraint supports the correct use of defaults, preventing more than one implementation from being designated the default implementation for a du. Knowledge of consistency constraints on standard attribute classes is built-in to the Library. Although the information expressed as customized attribute classes may also be subject to constraints. we have not considered the question of how such constraints may be expressed or enforced.

A final issue concerns the dynamic nature of names in our approach. Subject to authorization, users are permitted to create and modify names for implementations. Thus, an implementation's set of names may change over time. This mutability raises issues of consistency and repeatability. A user selecting an implementation twice with the same description may not retrieve the same implementation because some or all of the attributes he considered have changed. Similarly, two composition operations using the same description may produce compositions containing different implementations. One implication of this is that a composition may not be constructed using existing compositions as components, even if both composition operations use identical descriptions, because attribute values used to construct the existing composition may have changed. We do not understand the consequences of this ability to rename implementations and suggest further study to determine its effects.

Bibliography

1. J. L. Archibald. The External Structure. Tech. Rep. RC 8652, IBM Corporation, T. J. Watson Research Center, Yorktown Heights, New York, January, 1981.

2. J. L. Archibald. The External Structure: Experience with an Automated Module Interconnection Language. Proceedings of the ACM Software Engineering Symposium, June, 1981, pp. 147-157.

3. M. M. Astrahan and D. D. Chamberlin. Implementation of a Structured English Query Language. *Communications of the ACM 18*, 10 (October 1975).

4. M. M. Astrahan, et. al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems 1*, 2 (June 1976).

5. M. M. Astrahan, et. al. System R, A Relational Database Management System. *IEEE Computer Society: Computer 12*, 5 (May 1979).

6. T. Bloom. Dynamic Module Replacement in a Distributed Programming System. Ph.D. Th., Massachusetts Institute of Technology, Cambridge, Massachusetts, March, 1983.

7. T. A. Cargill. A View of Source Text for Diversely Configurable Software. Ph.D. Th., University of Waterloo, Waterloo, Ontario, 1979.

8. T. A. Cargill. Management of the Source Text of a Portable Operating System. Proceedings of the Fourth IEEE Computer Software and Applications Conference, 1980, pp. 764-68.

9. D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. Proceedings of the 1974 ACM SIGMOD Workshop on Data Description, Access and Control, 1974.

10. T. E. Cheatham. An Overview of the Harvard Program Development System. In *Software Engineering Environments*, H. Hunke, Ed., North-Holland Publishing Company, 1981, pp. 253-266.

11. T. E. Cheatham. Comparing Programming Support Environments. In *Software Engineering Environments*, H. Hunke, Ed., North-Holland Publishing Company, 1981, pp. 11-25.

12. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM 13*, 6 (June 1970).

13. L. W. Cooprider. The Representation of Families of Software Systems. Ph.D. Th., Carnegie-Mellon University, Pittsburgh, Pennsylvania, April, 1979.

14. E Cristofor, T. A Wendt, and B C Wonsiewicz Source Control + Tools = Stable Systems. Proceedings of the Fourth IEEE Computer Software and Applications Conference, 1980, pp. 527-532.

15. C. J. Date. *An Introduction to Database Systems* Addison-Wesley Publishing Company, 1981. Third Edition

16. F. DeRemer and H Kron Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976), 80-86.

17. R. S. Eanes, C. K. Hitchon, R. M. Thall, and J. W. Brackett. An Environment for Producing Well-Engineered Microcomputer Software. Proceedings of the Fourth International Conference on Software Engineering, September, 1979, pp. 386-398.

18. S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software Practice and Experience 9*, 4 (April 1979), 255-65.

19. N. Haberman, R. Ellison, R. Medina-Mora, P. Feiler, D. S. Notkin, G. E. Kaiser, D. B. Garlan, and S. Popovich. The Second Compendium of Gandalf Documentation. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May, 1982.

20. N. Haberman, D. Perry, P. Feiler, R. Medina-Mora, D. Notkin, G. Kaiser, R. Ellison, and D Garlan. A Compendium of Gandalf Documentation. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April, 1981.

21. K. E. Huff. A Database Model for Effective Configuration Management in the Programming Environment. Proceedings of the Fifth International Conference on Software Engineering, 1981, pp. 54-61.

22. E. L. Ivie. *The Programmer's Workbench - A Machine for Software Development. Communications of the ACM 20*, 10 (October 1977), 746-753.

23. B. W Lampson and E. E. Schmidt. Organizing Software in a Distributed Environment. Proceedings of the Sigplan '83 Symposium on Programming Language Issues in Software Systems, June, 1983, pp. 1-13.

24. B. W. Lampson and E. E. Schmidt. Practical Use of a Polymorphic Applicative Language. Proceedings of the Tenth Symposium on Principles of Programming Languages, January, 1983, pp. 237-255.

25. B. H Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler and A. Snyder. *Lecture Notes in Computer Science. Vol. 114: CLU Reference Manual.* Springer-Verlag, New York, 1981.

26. B. H. Liskov, A. Snyder, R. R. Atkinson, and J. C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM 20*, 8 (August 1977).

27. J. G. Mitchell, W. Maybury, and R. Sweet. Mesa Language Manual, Version 5.0. Tech. Rep. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, California, April, 1979.

28. D L Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM 15*, 12 (December 1972), 1053-1058.

29. *PDS User's Manual.* Ha-vard University Center for Research in Computing Technology. Aiken Computation Laboratory, Cambridge, Massachusetts, 1982.

30. M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering SE-1*, 4 (December 1975), 364-370.

31. E. E. Schmidt. Controlling Large Software Development in a Distributed Environment. Ph.D. Th., University of California at Berkeley, December, 1982.

32. *System Specification for the Ada Language System, Specification No. CR-CP-0059-A00.* Fort Monmouth, New Jersey, 1981. Contract No. DAAK80-80-C-0507.

33. J. Thomas. Module Interconnection in Programming Systems Supporting Abstraction. Ph.D. Th., Brown University, Providence, Rhode Island, 1976.

34. W. F. Tichy. Software Development Control Based on Module Interconnection. Proceedings of the Fourth International Conference on Software Engineering, September, 1979, pp. 29-41.

35. W. F. Tichy. Software Development Control Based on System Structure Description. Ph.D. Th., Carnegie-Mellon University, Pittsburgh, Pennsylvania, January, 1980.

36. *Unix Users Manual.* Bell Telephone Laboratories, Murray Hill, New Jersey, 1981. Release 4.0.

37. D. Weinreb and D. Moon. Lisp Machine Manual. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

OFFICIAL DISTRIBUTION LIST

1984

Director — 2 Copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research — 2 Copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 — 6 Copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center — 12 Copies
Cameron Station
Alexandria, VA 22314

National Science Foundation — 2 Copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 — 1 Copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

Dr. G. Hooper, USNR — 1 Copy
NAVDAC-OOH
Department of the Navy
Washington, DC 20374

-8

DTIC